



UNIVERSIDADE DO VALE DO TAQUARI – UNIVATES

CURSO DE SISTEMAS DE INFORMAÇÃO

**IMPLEMENTAÇÃO DE INFRAESTRUTURA COMO CÓDIGO PARA
PROVISIONAMENTO E *DEPLOY* DE APLICAÇÕES**

Jones Luís Noll

Lajeado, junho de 2020

Jones Luís Noll

IMPLEMENTAÇÃO DE INFRAESTRUTURA COMO CÓDIGO PARA PROVISIONAMENTO E *DEPLOY* DE APLICAÇÕES

Monografia apresentada na disciplina de Trabalho de Conclusão de Curso – Etapa II, do curso de Sistemas de Informação, da Universidade do Vale do Taquari – Univates, como parte da exigência para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Me. Fabrício Pretto

Lajeado, junho de 2020

RESUMO

A evolução tecnológica dos últimos anos trouxe uma necessidade crescente por softwares, devendo estes, estarem sempre disponíveis, acessíveis e atualizados para seus usuários. Para atender as demandas dos Sistemas de Informação, a infraestrutura passou a ser construída através de código, tornando-se assim, dinâmica e escalável. Ao utilizar os conceitos de DevOps e Infraestrutura como Código, este trabalho teve como objetivo a criação automatizada de toda a infraestrutura necessária para execução do ciclo de desenvolvimento de um software, desde o registro da mudança até sua entrega. Foram realizadas a criação e a configuração das máquinas virtuais e contêineres necessários, a orquestração dos mesmos em um *cluster* Kubernetes, finalizando com o *deploy* da aplicação e do banco de dados. Como resultado, após a execução de todos os processos, um sistema de informação encontrou-se em execução na estrutura e disponível aos usuários. Foram analisados os tempos necessários para a criação automatizada da estrutura e os benefícios na adoção da Infraestrutura como Código.

Palavras-chave: Infraestrutura. DevOps. Provisionamento. Automação. Deploy.

ABSTRACT

The technological evolution of the last few years has brought an increasing need for softwares, which should always be available, accessible and updated for its users. In order to meet the demands of Information Systems, the infrastructure started to be built through code, thus becoming dynamic and scalable. By using the concepts of DevOps and Infrastructure as Code, this work aimed at the automated creation of all the necessary infrastructure to execute the software development cycle, from the registration of the change until its delivery. The creation and configuration of the necessary virtual machines and containers were carried out, their orchestration in a Kubernetes cluster, ending with the deployment of the application and the database. As a result, after the execution of all the processes, an information system was found to be running in the structure and available to users. The time required for the automated creation of the structure and the benefits of adopting Infrastructure as a Code were analyzed.

Keywords: Infrastructure. DevOps. Provisioning. Automation. Deploy.

LISTA DE FIGURAS

Figura 1 - Muro da confusão	15
Figura 2 - Ciclo de vida do software	20
Figura 3 - Ciclo DevOps	21
Figura 4 - Visão geral da arquitetura desenvolvida	38
Figura 5 - Estrutura de arquivos do Terraform	39
Figura 6 - Terraform: arquivo main.tf	40
Figura 7 - Terraform: arquivo network.tf	41
Figura 8 - Terraform: arquivo vars.tf	42
Figura 9 - Terraform: arquivo vm_infra.tf	43
Figura 10 - Terraform: criação da VM infraestrutura	43
Figura 11 - VM infraestrutura em execução	44
Figura 12 - Estrutura de arquivos do Ansible	45
Figura 13 - Ansible: <i>Playbook</i> Aplicação	45
Figura 14 - Ansible: Role App.....	47
Figura 15 - Ansible: arquivo <i>hosts</i>	48
Figura 16 - Página inicial do Jenkins.....	49
Figura 17 - Jenkins: atualizar sistema de notas	50
Figura 18 - Jenkins: criar ou atualizar infraestrutura	51
Figura 19 - Aplicação: tela inicial.....	51
Figura 20 - Aplicação: listar notas	52
Figura 21 - Aplicação: cadastrar nota.....	53

LISTA DE QUADROS

Quadro 1 - Ferramentas utilizadas nos trabalhos relacionados	33
Quadro 2 - Ansible: Roles utilizadas	46
Quadro 3 - Tempos de execução para VM infraestrutura	54
Quadro 4 - Tarefas com maior tempo de execução para VM infraestrutura.....	55
Quadro 5 - Hardware de execução para VM infraestrutura.....	55
Quadro 6 - Tempos de execução para infraestrutura da aplicação.....	55
Quadro 7 - Tarefas com maior tempo de execução para infraestrutura da aplicação	56
Quadro 8 - Hardware de execução para infraestrutura da aplicação	56
Quadro 9 - Número de arquivos gerados	57

LISTA DE ABREVIATURAS E SIGLAS

CPU	<i>Central Processing Unit</i>
CRUD	<i>Create, Read, Update and Delete</i>
DNS	<i>Domain Name System</i>
EC2	<i>Elastic Compute Cloud</i>
GB	<i>Gigabyte</i>
IaC	<i>Infrastructure as Code</i>
IP	<i>Internet Protocol address</i>
RAM	<i>Random Access Memory</i>
SGBD	Sistemas de Gestão de Base de Dados
SSH	<i>Secure Shell</i>
TI	Tecnologia da Informação
VCPU	<i>Virtual CPU</i>
VM	<i>Virtual Machine</i>

SUMÁRIO

1 INTRODUÇÃO	9
1.1 Motivação.....	11
1.2 Objetivos	12
1.2.1 Objetivo geral	12
1.2.2 Objetivos específicos.....	12
1.3 Organização do trabalho	13
2 FUNDAMENTAÇÃO TEÓRICA	14
2.1 TI Tradicional	14
2.2 DevOps.....	16
2.2.1 Implementação	17
2.2.2 Entrega contínua	18
2.2.3 Ciclo de vida	20
2.3 Infraestrutura como código.....	22
2.3.1 Benefícios da infraestrutura como código.....	24
2.3.2 Princípios da infraestrutura como código	26
3 TRABALHOS RELACIONADOS	31
4 MATERIAIS E MÉTODOS	34
4.1 Tipo de pesquisa	34
4.1.1 Definição da pesquisa quanto aos seus objetivos.....	34
4.1.2 Definição da pesquisa quanto à natureza da abordagem.....	35
4.1.3 Definição da pesquisa quanto aos procedimentos técnicos	35
4.2 Tecnologias	36
4.3 Desenvolvimento.....	37
4.3.1 Provisionamento da infraestrutura em <i>cloud</i>	39
4.3.2 Gerenciamento de configurações.....	44
4.3.3 Automação com Jenkins	48
4.3.4 Aplicação para homologar infraestrutura	51
4.4 Testes e análise dos resultados	54
5 CONSIDERAÇÕES FINAIS	59

REFERÊNCIAS.....61

1 INTRODUÇÃO

A evolução tecnológica dos últimos anos nos levou à uma necessidade crescente por softwares, devendo estes, estarem sempre disponíveis e acessíveis aos usuários. Sommerville (2011) destaca que não haveria como a sociedade moderna funcionar sem a existência de complexos sistemas de software, pois os mesmos são essenciais para que serviços como energia, comunicação e transporte, tenham seu funcionamento adequado. O autor acrescenta que os sistemas de software permitiram a exploração do espaço e a criação do mais importante sistema de informação de todos os tempos, a Internet.

O mercado cada vez mais competitivo, tem colocado as empresas em uma corrida pelo lançamento de novas funcionalidades de software, disponíveis para diferentes dispositivos e sistemas operacionais, integrando-os a outros serviços. Isso não só aumentou a complexidade do processo de desenvolvimento de software, como também da infraestrutura, dificultando que novas versões sejam aplicadas em produção de maneira eficaz e eficiente.

Outro fator que contribui para a ineficiência na evolução do software, diz respeito às diferentes habilidades, vocabulários e experiências entre as equipes de operações e desenvolvimento de software. Além disso, tais equipes possuem objetivos diferentes, onde enquanto o desenvolvimento objetiva a evolução do software, voltado ao lançamento de novas funcionalidades, as equipes de operações objetivam a estabilidade, sendo completamente resistentes a qualquer mudança, ocasionando conflito entre as equipes (BEYER et al., 2016).

DevOps nasceu para resolver o conflito entre as equipes de desenvolvimento e operações, de modo que consigam trabalhar em conjunto, possuindo os mesmos objetivos e responsabilidades. A Amazon (2019), definiu DevOps como uma combinação de filosofias culturais, práticas e ferramentas que objetivam aumentar a capacidade de uma organização entregar software e serviços, aumentando assim a qualidade de atendimento aos seus clientes e a competitividade.

Entre os pilares da metodologia DevOps estão a integração e entrega contínua, onde o software vive em constante evolução e novas funcionalidades são disponibilizadas com velocidade aos usuários. Vadapalli (2018) define integração contínua como o trabalho em conjunto de diversos desenvolvedores, integrando seus códigos com o auxílio de um repositório. Entre os benefícios, o autor cita o rápido acesso ao código mais recente e a redução de erros em códigos. A entrega contínua, é definida pelo autor como uma entrega rápida e automatizada de software repetitivamente, possuindo como principais benefícios a automação e a confiabilidade em que o mesmo é disponibilizado.

DevOps utiliza muita automação, inclusive na infraestrutura, criando o conceito de *Infrastructure as Code* (IaC), que em tradução livre, significa Infraestrutura como Código. A Hewlett Packard Enterprise (2019) definiu que Infraestrutura como Código trata a infraestrutura da mesma forma como um software programável, onde toda a infraestrutura necessária para a execução de um software pode ser criada através da execução de um código previamente desenvolvido. Riti (2018), acrescenta que esta tecnologia permite a criação de ambientes exatamente iguais, tornando possível que a equipe de desenvolvimento tenha um ambiente idêntico ao de produção para desenvolver e testar seu software. Desta forma, é possível reduzir de forma drástica os tempos para liberação de um software, e além disso, reduzir os erros por testes em ambiente heterogêneos.

A infraestrutura como código ajuda a resolver problemas de escalabilidade e gerenciamento, tendo como principais benefícios: o controle de versão do código, podendo facilmente avançar ou retroceder para um estado conhecido; a escalabilidade, tornando possível duplicar ambientes de forma rápida e segura; e a recuperação de desastres, onde como a infraestrutura toda é baseada em código,

torna possível restaurar o ambiente através da execução do código existente em backup (NELSON-SMITH, 2013).

Empresas gigantes, como Amazon, Netflix, Google e Facebook, já utilizam Infraestrutura como Código para prover seus serviços, o que prova a eficiência e segurança desta prática. Nestas empresas, os sistemas não são apenas críticos para o negócio, eles são o negócio, não havendo tolerância para indisponibilidades (MORRIS, 2016).

1.1 Motivação

O avanço nas tecnologias de desenvolvimento de software, combinadas com o modelo ágil de desenvolvimento, contribuíram para que softwares fossem desenvolvidos com maior velocidade e eficiência. Entretanto, isso acabou agravando ainda mais os conflitos existentes entre as equipes de desenvolvimento e operações, principalmente devido a infraestrutura não acompanhar este ritmo, sendo ainda configurada de forma manual, trabalhosa e sem homogeneidade. Essa falta de automatização e padrão de ambiente, acaba fazendo com que ambientes de homologação sejam criados com grande esforço da equipe de operação, porém muitas vezes diferentes do ambiente de produção, gerando problemas no *deploy*¹ dos softwares.

A complexidade das infraestruturas atuais e a falta de documentação, acabam gerando problemas de continuidade para as empresas, pois em uma eventual falha, o tempo de recuperação tende a ser grande e penoso. Além disso, são necessárias soluções de recuperação de desastre, muitas vezes complexas e com custos inviáveis. Kim et al. (2018), acrescentam que a infraestrutura tradicional é complexa, mal documentada e frágil, onde normalmente são realizadas soluções paliativas aos problemas encontrados, deixando para um futuro que nunca chega, as soluções definitivas.

¹ *Deploy* quando relacionado a software, significa implantá-lo ou colocá-lo no ar.

1.2 Objetivos

Nesta seção são apresentados os objetivos gerais e específicos do projeto proposto.

1.2.1 Objetivo geral

O objetivo geral deste trabalho é implementar um ambiente de homologação e produção de um produto de software, fazendo uso dos conceitos e técnicas de Infraestrutura como Código.

1.2.2 Objetivos específicos

Os objetivos específicos são:

- a. Pesquisar e conhecer os conceitos de DevOps e IaC;
- b. Modelar a arquitetura da infraestrutura a ser construída;
- c. Implantar ferramentas que tornem possível o provisionamento, gerenciamento de configurações e *deploy* de servidores;
- d. Manter o código da infraestrutura versionado;
- e. Automatizar o processo de *deploy* de software;
- f. Realizar a integração entre as ferramentas utilizadas;
- g. Analisar os resultados obtidos.

1.3 Organização do trabalho

Com o objetivo de auxiliar na compreensão deste trabalho, o mesmo foi dividido em capítulos, sendo estes divididos da seguinte forma:

No capítulo 2 encontra-se a fundamentação teórica, utilizada como base no desenvolvimento do trabalho, iniciando com os conceitos e características de DevOps, passando pelas características, conceitos, benefícios e princípios da Infraestrutura como Código.

No capítulo 3 são documentados estudos semelhantes ao deste trabalho, utilizados como apoio e validação do tema.

O capítulo 4 é formado pela metodologia utilizada e o método científico que o presente trabalho se enquadra. Além disso, é apresentada a implementação da proposta do trabalho e os resultados obtidos.

No capítulo 5 são expostas as considerações finais.

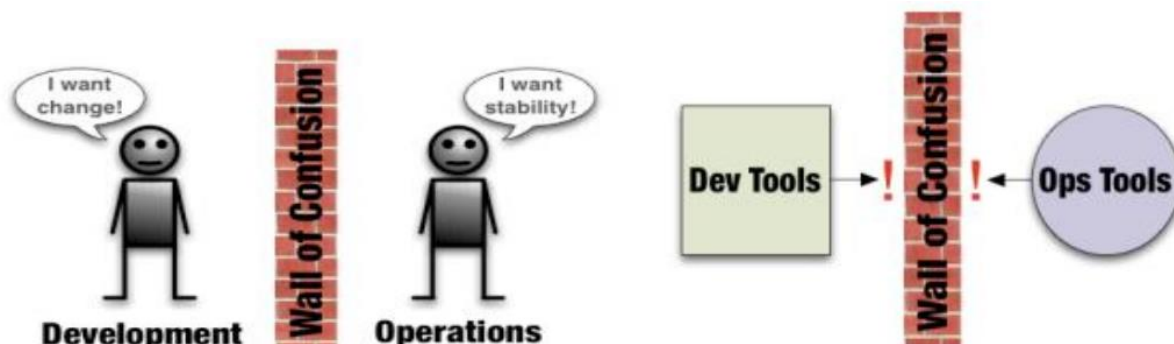
2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será apresentado o embasamento teórico, coletado de referências literárias e utilizado para teorizar e fundamentar a temática deste trabalho. São abordados os conceitos de DevOps e entrega contínua, a necessidade de aproximar as equipes de desenvolvimento e operações, além do conceito, dos benefícios e dos princípios da infraestrutura como código.

2.1 TI Tradicional

No modelo tradicional, muitos departamentos de TI possuem os times de desenvolvimento e operações com clara divisão de responsabilidades. Enquanto um é responsável por criar, manter e evoluir softwares, o outro é responsável por mantê-los em operação. Isso acaba por gerar conflitos entre as equipes, pois enquanto o time de desenvolvimento é motivado a realizar mudanças em suas aplicações, o que pode gerar certa instabilidade, o time de operações é motivado pela estabilidade, evitando mudanças que podem trazer risco (SATO, 2014). Shafer (2009) complementa denominando os conflitos entre as equipes de operação e desenvolvimento de “*Wall of Confusion*”, que em tradução livre significa muro da confusão, conforme apresentado na Figura 1.

Figura 1 - Muro da confusão



Fonte: Edwards (2010, texto digital).

A Figura 1 exemplifica o muro existente entre as equipes, que as divide e as mantém em constante conflito, principalmente devido às mesmas possuírem objetivos diferentes, uma objetivando as mudanças e a outra a estabilidade (EDWARDS, 2010).

Segundo Kim et al. (2018), este conflito entre os times acaba impedindo que os objetivos globais da organização sejam atendidos, e além disso, faz com que os profissionais de TI façam suas entregas de software e serviço com baixa qualidade, trazendo uma experiência ruim aos seus clientes.

Ainda segundo Kim et al. (2018), são três os fatores principais que levam os departamentos de TI a uma decadência de eficiência. Estes fatores, segundo os autores, devem ser conhecidos pela maioria dos profissionais de TI, e ocorrem em empresas de todos os tamanhos:

- a) Infraestrutura complexa, mal documentada e frágil: isso leva a soluções de contornos diárias, deixando para um futuro que nunca chega, as soluções definitivas;
- b) Compensação de uma promessa não cumprida: isso ocorre quando, por exemplo, um gerente de produto promete um novo recurso para compensar um atraso de entrega ou *bug*. Isso acaba alterando as prioridades das equipes, que para atender a nova demanda, acabam realizando seus trabalhos aceleradamente, não seguindo as melhores práticas;
- c) Tudo fica um pouco mais difícil: devido aos fatores acima, pouco a pouco, todos os profissionais ficam mais ocupados, as tarefas e a comunicação

ficam mais lentas, e as filas de pendências só aumentam. Isso faz com que atividades pequenas causem grandes falhas, deixando as equipes mais apreensivas e intolerantes a fazer mudanças.

Sato (2014) comenta que esta decadência nos departamentos de TI gera uma burocracia no processo de colocar códigos de desenvolvimento e teste para produção. Com o tempo, a burocracia só aumenta, diminuindo a frequência de *deploys*. Como a frequência diminui, alterações de código são acumuladas, tornando os *deploys* cada vez mais perigosos.

2.2 DevOps

Sharma (2014), define DevOps como a junção abreviada de desenvolvimento e operações, sendo basicamente uma metodologia de trabalho baseada no modelo ágil, onde todos os envolvidos trabalham para realizar a entrega do software de maneira contínua. Hüttermann (2012) agrupa na equipe de desenvolvimento os programadores e analistas de software, testadores e a equipe de garantia de qualidade. Já na equipe de operações, o autor agrupa os administradores de sistema, administradores de banco de dados e técnicos de rede.

DevOps se caracteriza pelo envolvimento da TI em todas as fases do design e desenvolvimento de uma aplicação, substituindo interações humanas por forte automação, e utilizando de metodologias e ferramentas em atividades de operação (BEYER et al., 2016). Hüttermann (2012) acrescenta que DevOps cria uma cultura de compartilhamento, onde as pessoas compartilham ideias, processos e ferramentas. Na cultura DevOps, as pessoas estão sempre antes de processos e ferramentas, onde o software é feito por e para pessoas.

DevOps tenta resolver os conflitos entre as equipes de desenvolvimento e operações, criando uma cultura de colaboração entre profissionais de habilidades complementares, trabalhando como parte da mesma equipe (SATO, 2014). Para Geerling (2015), quando as equipes de operação e desenvolvimento conseguem trabalhar juntas, acaba-se ganhando velocidade no desenvolvimento, gastando menos

tempo resolvendo problemas, e gerando tempo para atividades como ajuste de desempenho e testes.

Para Kim et al. (2018), o movimento DevOps iniciou-se em 2008 durante a conferência Ágil, em Toronto no Canadá. Neste evento, Patrick Debois e Andrew Schafer realizaram uma apresentação informal sobre a aplicação da metodologia Ágil na infraestrutura. Em 2009, durante a conferência Velocity, John Allspaw e Paul Hammond apresentaram como a equipe de desenvolvimento e operações trabalhavam em conjunto no Flickr², com objetivos compartilhados e integração contínua, fazendo da implementação parte das tarefas diárias de todos, podendo chegar a até dez *deploys* por dia. Sato (2014) acrescenta que o aumento de deploys também reduz o risco associado ao mesmo. Como os *deploys* ocorrem com frequência, o número de alterações em cada *deploy* diminui, agilizando o trabalho de encontrar o motivo do problema quando algo dar errado.

2.2.1 Implementação

Riti (2018) aponta que a adoção de DevOps em uma empresa é o início de uma grande jornada, onde o gerenciamento assume papel fundamental, devendo iniciar as mudanças necessárias e integrando-as à cultura da empresa. Para que o processo seja bem sucedido, devem ser tomadas algumas ações:

- a) A gerência deve promover a mudança;
- b) O responsável pelo software deve ser o desenvolvedor;
- c) A equipe de operações deve ter tratamento especial;
- d) Construção de políticas de integração e entrega contínuas;
- e) O departamento de TI deve ter suas barreiras removidas;
- f) Automatização do processo de liberação de software;

² Site de hospedagem e compartilhamento de imagens, cujo o endereço é <https://www.flickr.com>.

g) Práticas ágeis devem ser promovidas por toda a empresa.

Para Vadapalli (2018) as organizações devem considerar a implementação de DevOps quando necessitarem atender a alguns objetivos:

- a) Automatizar a infraestrutura e configurar um fluxo de trabalho;
- b) Automatizar repositórios de códigos, compilações e testes;
- c) Implementar a integração e entrega contínuas;
- d) Implementar a virtualização, containerização e balanceamento de carga;
- e) Projetos de *big data* e mídia social;
- f) Projetos de *machine-learning*, que em tradução livre, significa aprendizagem de máquina.

2.2.2 Entrega contínua

Entrega contínua, segundo Kim et al. (2018), é garantir que código e a infraestrutura estejam alinhados, e em um estado implementável. Dessa forma, qualquer código pode passar para produção de forma segura. Para isso, é utilizado *build*, teste e integração contínuos.

Sato (2014) descreve *build* como um processo que envolve todas as atividades necessárias para conseguir executar um software. Entre estas atividades, estão a compilação, resolução de dependências, vinculação com bibliotecas e empacotamento. No final do processo, como resultado, é gerado um arquivo binário chamado artefato. Com o crescimento de um sistema, o processo de *build* acaba ficando mais complicado, dificultando que todos os passos sejam lembrados e executados manualmente, e por isso, é importante investir na automação.

Segundo Kim et al. (2018), testes contínuos e automatizados levam a correções de problemas em minutos, uma vez que os erros são descobertos rapidamente. Isso também leva a um aprendizado contínuo da equipe, pois a raiz do problema é

encontrada de forma rápida, o que evita possíveis esquecimentos com soluções postergadas. Sato (2014) descreve que existem diversos tipos de testes com possibilidade de automatização, e entre eles, cita aqueles que para o autor são os mais comuns:

- a) Testes de unidade: executam partes de código isoladamente, sem que haja necessidade de executar a aplicação como um todo. Como executam isoladamente e são muito rápidos, podem ser executados frequentemente;
- b) Testes de integração ou componente: executam uma parte da aplicação com a finalidade de verificar como ela se integra com suas dependências;
- c) Testes funcionais ou de aceitação: realizam testes da aplicação como um todo, simulando o uso de um usuário.

A integração contínua consiste em diversos desenvolvedores de software trabalhando em conjunto, integrando continuamente seus códigos a um repositório compartilhado entre a equipe (VADAPALLI, 2018). Sato (2014) acrescenta que as equipes necessitam de disciplina no envio do código para o repositório, para que ao concluir uma tarefa, suas mudanças funcionem corretamente com o restante do software. Entre os benefícios desta prática estão a agilidade na disponibilidade do código, ciclos de compilação mais rápidos e transparência no processo de construção do software (VADAPALLI, 2018).

A entrega contínua permite o desenvolvimento rápido e confiável do software, levando a uma entrega do produto de forma automatizada, com o mínimo de esforço manual. Entre os benefícios da entrega contínua estão a máxima automação do processo, e a rapidez e confiabilidade que o software é implementado repetitivamente (VADAPALLI, 2018).

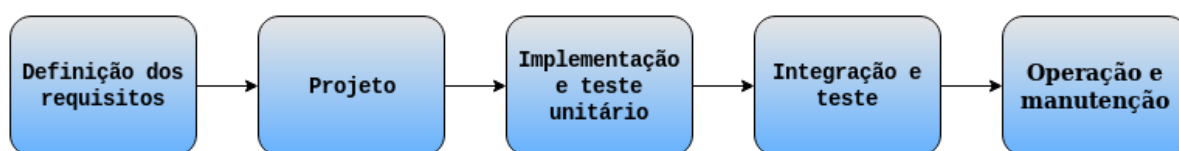
Com o conceito de entrega contínua, várias empresas de sucesso, como Google, Amazon, Netflix e Facebook, conseguem rapidamente colocar uma nova idéia em produção, tornando-se competitivas e adaptáveis às mudanças de mercado (SATO, 2014). Para Sharma (2014) a entrega contínua de software faz com que as empresas aproveitem melhor as oportunidades, e além disso, reduz consideravelmente o tempo de retorno para seus clientes.

Kim et al. (2018), citam que nas décadas de 70 e 80, a implementação de novos recursos poderiam levar de um a cinco anos, do desenvolvimento à implementação. Em 2000, com a adoção de novas metodologias de desenvolvimento e o avanço tecnológico, o tempo de desenvolvimento caiu para semanas ou meses, entretanto a sua implementação em produção ainda levava muito tempo e era recheada de problemas. A partir de 2010, com a utilização de DevOps e recursos de virtualização e nuvem, aplicações inteiras e até mesmo empresas, foram criadas e implementadas em produção em semanas. Isso trouxe a capacidade das organizações de testar seus produtos de maneira rápida e com baixo custo, descobrindo com rapidez quais são mais aderentes ao mercado.

2.2.3 Ciclo de vida

O processo clássico de desenvolvimento de software é caracterizado pelo encadeamento entre uma fase e outra, sendo estas fases conhecidas como definição dos requisitos, projeto, implementação e teste unitário, integração e teste, e por fim, operação e manutenção, conforme exibido na Figura 2 (SOMMERVILLE, 2011).

Figura 2 - Ciclo de vida do software



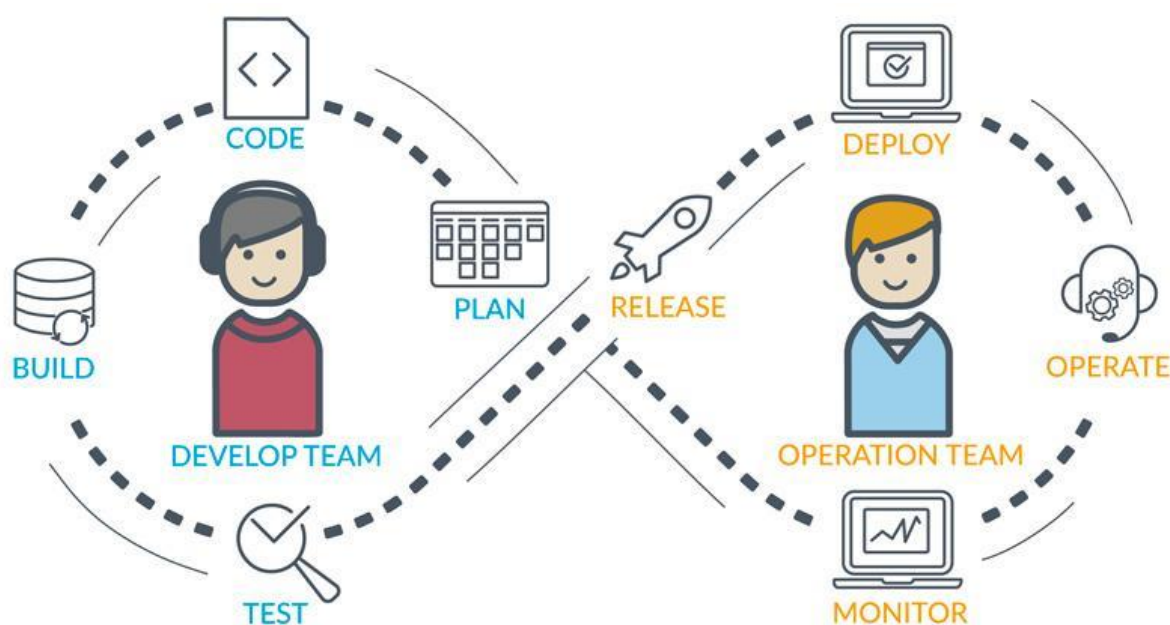
Fonte: Elaborado pelo autor, baseado em Sommerville (2011).

No ciclo de vida do software, exibido na Figura 2, primeiramente são realizadas atividades de análise e definição de requisitos, coletando junto aos usuários os objetivos e características do sistema, para na sequência definir os detalhes e especificações. Na próxima fase é realizado o projeto do sistema, que define a arquitetura geral e tudo que será necessário para seu funcionamento. Na fase de implementação, o sistema é desenvolvido e testado de forma unitária e em módulos, sendo estes integrados e testados como um sistema completo na etapa de integração e testes. A última fase é a de operação e manutenção, onde é realizada a instalação

e liberação para o uso do sistema. A manutenção consiste na correção de todos os erros que não foram descobertos nas etapas iniciais (SOMMERVILLE, 2011).

Enquanto isso, Humble e Farley (2010), mostram que o ciclo de vida DevOps tem como base o processo de integração e entrega contínua, possuindo três objetivos principais: primeiro, torna visível para todos os envolvidos cada parte do processo de construção, implantação, teste e entrega de software. Segundo, otimiza o *feedback*, tornando possível a identificação e resolução dos problemas com brevidade. O terceiro, através de um processo completamente automatizado, permite que sejam liberadas a vontade qualquer versão de software para qualquer ambiente. Este ciclo é exibido na Figura 3.

Figura 3 - Ciclo DevOps



Fonte: Ventapane (2019, texto digital).

A Figura 3 mostra que o ciclo de vida DevOps também possui as fases do desenvolvimento de software, entretanto com integração contínua entre as fases e muita automação de processos (HUMBLE; FARLEY, 2010).

2.3 Infraestrutura como código

O gerenciamento e implementação de servidores com confiabilidade e eficiência, tem sido desafiador desde a introdução das redes de computadores. Instalar software, alterar configurações e administrar serviços, eram feitos de forma manual e individual em cada servidor. Com o aumento de complexidade das aplicações, e como consequência, dos *datacenters*, ficou claro que não seria possível escalar este gerenciamento manual na mesma velocidade que as aplicações, e por isso, surgiram as ferramentas de provisionamento e configuração de servidores (GEERLING, 2015).

Para Nelson-Smith (2013), a infraestrutura como código teve início em 2006, quando a Amazon *Web Services* lançou o *Elastic Compute Cloud* (EC2). A partir daquele momento, qualquer ideia para uma aplicação *web* pode ser implementada em questão de semanas, fazendo com que pequenas empresas tivessem um crescimento acelerado, e mesmo sendo lideradas por desenvolvedores, tiveram que se preocupar com rotinas de equipes de operações, como adicionar máquinas idênticas, fazer *backup*, escalar banco de dados, entre outras. Como estas equipes eram pequenas e tinham que administrar ambientes complexos, começaram a surgir as primeiras ferramentas de gestão de configuração.

Morris (2016), coloca a criação da nuvem computacional como um divisor de águas, criando a idade do ferro e a idade da nuvem. Na idade do ferro todos os sistemas estavam apoiados ao hardware físico, e com muito trabalho manual para provisionar e manter a infraestrutura. Na idade da nuvem não há mais uma associação dos sistemas com o hardware físico, e grande parte do trabalho de provisionamento e manutenção da infraestrutura podem ser realizados através de software, com um ganho de tempo significativo para que os profissionais consigam realizar outras atividades.

Segundo Geerling (2015), foi a virtualização de servidores que inaugurou o gerenciamento de infraestrutura em larga escala. Ao invés de realizar diversos trabalhos de forma manual, os administradores passaram a criar novos servidores automaticamente, ou com o mínimo de intervenção. Com isso, cresceu consideravelmente a capacidade de gerenciamento de servidores dos

administradores, conseguindo administrar um número muito maior de servidores no mesmo tempo.

Infraestrutura como código pode ser definida como o uso das práticas do desenvolvimento de software para a automação da infraestrutura, utilizando-se de rotinas consistentes e repetíveis, possibilitando a alteração de sistemas e suas configurações, através de sistemas com validações (MORRIS, 2016). Artač et al. (2017), definem infraestrutura como código como o uso de código fonte para projetos de infraestrutura, sendo que todo o conjunto de *scripts*, códigos de automação e configuração, modelos, dependências necessárias e parâmetros de configuração sejam expressados com a utilização do mesmo. Hüttermann (2012) acrescenta que o objetivo da infraestrutura como código é lidar com as suas configurações da mesma forma como se lida com o desenvolvimento de um sistema, escolhendo a linguagem ou ferramenta adequada para iniciar o desenvolvimento de uma solução que atenda às necessidades, que seja executável e que possa ser aplicada de forma eficiente e repetitiva.

Morris (2016) lembra que a infraestrutura como código ganhou força com o surgimento da computação em nuvem, mas que boa parte dos princípios e práticas podem também serem aplicados a estruturas fora da nuvem, como ambientes virtualizados ou até mesmo em hardware físico. Segundo o autor, as plataformas de virtualização podem ser configuradas para possuir a capacidade de criação e destruição de servidores programaticamente como a nuvem, e o hardware físico apesar de ser provisionado manualmente, pode ser configurado e atualizado utilizando ferramentas de configuração baseadas em IaC.

Segundo a Instruct (2017), a infraestrutura como código é baseada na automação, sendo composta por três atividades principais:

- a) Gerência de configuração: utilização de ferramentas para manter a infraestrutura padronizada. A principal característica destas ferramentas é a gerência de estados do sistema. Ao encontrar uma diferença entre o estado declarado e o estado atual do sistema, as configurações serão corrigidas de forma automática. Saito, Lee e Wu (2017) acrescentam que com o gerenciamento de configuração é possível manter os arquivos de

configuração em um repositório de código, tornando possível o rastreamento do histórico de alterações realizadas na configuração do ambiente;

- b) Orquestração: consiste em um conjunto de instruções a serem executadas para cumprir um objetivo, em tempo real e com a possibilidade de ser paralela. Para Saito, Lee e Wu (2017), apesar de também ser uma ferramenta de gerenciamento de configuração, a ferramenta de orquestração acaba sendo mais inteligente e dinâmica ao configurar e alocar os recursos da nuvem. Isso ocorre por sua capacidade de gerenciar vários recursos do servidor, sendo capaz de implantar e configurar um aplicativo e a rede automaticamente, escolhendo ainda, o servidor que estiver disponível;
- c) Provisionamento: consiste em uma camada entre os recursos disponíveis e a necessidade. Basicamente é responsável por criar novos ambientes, de maneira rápida e fácil. Sato (2014) acrescenta que o provisionamento é uma das etapas iniciais de configuração de um novo recurso. O provisionamento de servidores, por exemplo, engloba todas as etapas para que o mesmo fique pronto para uso.

2.3.1 Benefícios da infraestrutura como código

Um dos grandes benefícios na utilização de infraestrutura como código, é a facilidade de replicação do ambiente. No modelo tradicional, este procedimento sempre poderá contar com um erro humano ou um erro de operação, já com a automatização, uma ferramenta de gerenciamento de configuração poderá criar o ambiente de forma automática e rápida (SAITO; LEE; WU, 2017).

Nelson-Smith (2013) cita benefícios que a infraestrutura como código traz, e que ajudam a resolver problemas de escalabilidade e gerenciamento:

- a) Repetibilidade: a construção da infraestrutura em uma linguagem de programação de alto nível traz mais confiança em relação a infraestrutura tradicional, pois a torna ordenada e repetível. O mesmo código deve produzir

sempre a mesma saída, levando a certeza que seja possível recriar o ambiente;

- b) Automação: implementar aplicativos com uso de ferramentas maduras, escritas em linguagens modernas, faz com que o próprio fato de abstrair infraestruturas gere os benefícios da automação. Sato (2014) acrescenta que investir em automação resulta na execução mais rápida das tarefas, além de diminuir a possibilidade de erros humanos;
- c) Agilidade: o gerenciamento de código fonte e controle de versão trazem a possibilidade de avançar ou retroceder para um estado conhecido. Desta forma, somos capazes de realizar mudanças drásticas na topologia com facilidade, respondendo com velocidade às mudanças impostas pelo negócio. Caso algo der errado, é possível analisar o histórico de mudanças, e ainda, como toda a infraestrutura é texto, comparar as mudanças que houveram entre as versões. Kim et al. (2018), acrescentam que realizar o controle de versão para o código de infraestrutura é tão importante quanto para o código das aplicações. Uma vez que na infraestrutura existem muito mais ajustes configuráveis do que nas aplicações;
- d) Escalabilidade: a repetibilidade e a automação trazem a possibilidade de aumentar a quantidade de servidores facilmente, principalmente quando o ambiente possui os benefícios do provisionamento rápido de hardware da nuvem. Além disso, como o código é modular, é possível realizar o reaproveitamento de código conforme a estrutura for crescendo;
- e) Reafirmação: todos os benefícios trazem segurança, principalmente pelo fato da arquitetura e o design da infraestrutura serem modelados, e não apenas implementados. Mas ter o código-fonte da infraestrutura significa ter uma base de conhecimento razoável, podendo visualizar como os sistemas funcionam e não correndo o risco de apenas um profissional ter o entendimento completo do ambiente;
- f) Recuperação de desastres: como toda a infraestrutura é escrita como código, caso ocorra um evento catastrófico que destrua todo o ambiente de produção, será possível facilmente restaurar o ambiente a partir do backup,

reimplementando toda a infraestrutura. Além disso, a infraestrutura como código facilita procedimentos de testes de recuperação.

Riti (2018) ainda cita como benefícios a facilidade de suporte, uma vez que o código está documentado e é possível retornar a um ponto funcional da infraestrutura, e a melhoria contínua, onde são seguidos os mesmos princípios que os utilizados para software, qualquer alteração é preparada, testada e finalmente lançada em produção.

Outro benefício é o fato da equipe de TI utilizar suas habilidades em coisas que agregam valor ao negócio, e não em tarefas rotineiras e repetitivas. Além disso, há possibilidade dos usuários provisionarem e gerenciarem os recursos da qual necessitam, sem a necessidade de intervenção da equipe de TI (MORRIS, 2016).

2.3.2 Princípios da infraestrutura como código

Nelson-Smith (2013) lembra que codificar infraestruturas complexas é uma tarefa desafiadora e cita os problemas mais comuns enfrentados por empresas ao colocar suas ideias em prática sem aplicar todos os princípios da infraestrutura como código:

- a) Códigos de infraestrutura espalhados;
- b) Duplicação de código e falta de compreensão do que ele faz;
- c) Código que começou bem projetado e testado, mas que acaba repleto de alterações desorganizadas e sem planejamento;
- d) A sensação de fim da organização se uma ou duas pessoas chaves saíssem, apesar de toda a base de conhecimento da infraestrutura que está no próprio código;
- e) Contos de que mudanças triviais tiveram efeitos colaterais catastróficos em outros lugares do ambiente.

Morris (2016) relaciona, aqueles que para o autor, são os princípios da infraestrutura como código:

- a) Ambientes com facilidade de reprodução: qualquer elemento da infraestrutura deve ser construído sem que sejam tomadas decisões significativas sobre como construir. Com os serviços e ambientes podendo ser provisionados com pouco esforço, grande parte dos riscos são removidos, gerando maior confiança para realizar alterações. As ferramentas que provisionam a infraestrutura devem fornecer as decisões sobre quais softwares instalar, nome do *host*, entre outras configurações;
- b) Ambientes são descartáveis: a infraestrutura deve ser dinâmica, com os sistemas assumindo que a infraestrutura sempre estará mudando. Os recursos devem ser facilmente criados, destruídos, substituídos, movidos e redimensionados. As ferramentas devem controlar e continuar a execução mesmo que servidores desapareçam, apareçam ou sejam redimensionados. Melhorias e correções podem ser realizadas com a infraestrutura em execução, devido a capacidade de lidar com as mudanças, o que consequentemente, torna os serviços com uma tolerância maior a falhas;
- c) Consistência: elementos da infraestrutura que exercem mesma função, fornecendo serviços semelhantes, devem ser praticamente idênticos, sendo diferenciados apenas por configurações que os identificam, como por exemplo, seu endereço *IP*. A possibilidade de criar e reconstruir a infraestrutura utilizando ferramentas de configuração, garante que todos os elementos de mesma função sejam criados iguais, mas tão importante quanto isso, é garantir que as mudanças que ocorrem após a criação destes sejam tratadas e inseridas como parte do código da infraestrutura. Ao infiltrar na infraestrutura, as inconsistências impedem que se tenha confiança na automação. Há duas maneiras de tratar as inconsistências: A primeira é definir a alteração para todos os elementos, e a segunda é criar uma nova classe criando um novo tipo de elemento com as diferenças propostas;
- d) Repetição de processos: com o uso de scripts e ferramentas de gerenciamento de configuração, deve haver a possibilidade de repetição de

qualquer ação que seja executada na infraestrutura. O uso destas ferramentas e *scripts* muitas vezes acaba gerando mais trabalho que realizar a alteração manualmente, porém ao realizar a atividade de forma manual, perde-se a possibilidade de repetibilidade. Como cultura de equipes de infraestrutura eficazes, toda tarefa que possa ser um *script* deve se tornar um, e caso um *script* não atenda, devem ser buscadas técnicas ou ferramentas que possam ajudar, ou então, verificar a possibilidade de abordar o problema de uma forma diferente;

- e) *Design* em constante mudança: na idade do ferro, como era impossível prever como um sistema se comportaria na prática, eram criados ambientes abrangentes e complexos, dificultando a alteração e a melhoria do mesmo. Além disso, qualquer alteração possuía um custo elevado. Já na idade da nuvem, com uma infraestrutura dinâmica, realizar alterações no ambiente existente tende a ser fácil e com baixo custo. Para isso, é importante que o ambiente seja projetado objetivando as mudanças, com o software e a infraestrutura atendendo os requisitos da maneira mais simples possível. Realizar alterações com frequência é a melhor forma de garantir que o ambiente seja alterado com segurança e rapidez, forçando que todos os profissionais envolvidos façam uso de boas práticas para gerenciar as mudanças e desenvolver processos simplificados e eficientes.

Nelson-Smith (2013) acrescenta princípios da infraestrutura como código, que segundo o autor, devem ser aplicados a todas as fases do processo de desenvolvimento da infraestrutura:

- a) Modularidade: serviços devem ser pequenos e simples;
- b) Cooperação: a sobreposição de serviços deve ser desencorajada pelo design, que deve cativar outras pessoas e serviços a realizar um consumo dos serviços existentes, promovendo uma melhoria contínua no *design* e implementação;
- c) Composto: a construção de sistemas completos e complexos deve ser realizada pela integração de serviços, como tijolos de construção;

- d) Extensibilidade: a modificação, aprimoramento e melhoria de serviços devem ser realizadas em resposta a novas necessidades;
- e) Flexibilidade: os serviços devem ser construídos em ferramentas que possam garantir a capacidade de resolver até os problemas mais complicados;
- f) Declaração: as especificações dos serviços devem levar em conta o que se quer fazer, e não como fazer;
- g) Abstração: deve-se pensar no componente e em sua função, e não com os detalhes da implementação;
- h) Idempotência: a configuração dos serviços deve ocorrer apenas quando necessário, e apenas uma vez. Sato (2014) acrescenta que esta é uma característica importante das ferramentas de gerenciamento de configuração, pois assegura que o mesmo código possa ser executado diversas vezes no mesmo *host*, alterando apenas o que for necessário;
- i) Convergência: os serviços devem estar com seu estado alinhado às suas políticas de configuração, convergindo para o funcionamento de um sistema geral.

Os autores supracitados definiram os princípios ligados a IaC, Riti (2018) cita as práticas que devem ser seguidas para conduzir a implementação com sucesso dos princípios da infraestrutura como código:

- a) Definir a infraestrutura em um arquivo de código: definir em um arquivo tudo o que for necessário para a infraestrutura, como por exemplo, DNS, espaço em disco, sistema operacional, entre outros. Somente com todas as configurações será possível tornar a infraestrutura programática e repetitiva;
- b) Documentar o sistema e o processo: uma boa documentação facilita a criação de melhorias e correções de problemas, mantendo a infraestrutura saudável ao longo do tempo;
- c) Versionar o código: o versionamento torna possível e facilita a recuperação do ambiente para qualquer estado definido, aumentando a consistência e

disponibilidade. Além disso, é importante para recuperar a infraestrutura em um eventual erro ou problema que destrua o ambiente;

- d) Testar: como toda a infraestrutura está definida em um arquivo de código, é possível testá-la antes de colocá-la em produção;
- e) Realizar pequenas alterações: pequenas alterações facilitam o isolamento da causa de um problema, facilitando sua correção.

3 TRABALHOS RELACIONADOS

Nesta seção, serão elencados alguns trabalhos relacionados ao tema deste estudo.

Rodrigues (2017) utilizou a ferramenta Puppet para o gerenciamento de configurações de servidores. A instituição onde o trabalho foi aplicado possui sua sede em Brasília, e diversas unidades espalhadas pelo território nacional, com mais de trinta servidores físicos e trezentos virtuais em sua estrutura, divididos entre produção, teste, desenvolvimento e homologação.

Entre os problemas encontrados, anteriores a aplicação da ferramenta, o autor cita a falta de instalação de alguns serviços em servidores, configurações de usuários inconsistentes e serviços essenciais fora da inicialização do sistema operacional. Com a utilização do Puppet, todos os problemas citados foram resolvidos, viabilizando ainda uma padronização contínua do ambiente.

Camões e Silva (2018) realizaram um estudo de caso sobre a utilização do Ansible como ferramenta de implantação no Tribunal de Justiça do Distrito Federal. A equipe de operações ao receber as requisições de novas máquinas virtuais, as criava e configurava utilizando scripts, com falhas de padronização. Além disso, atrasos de entrega de infraestrutura eram causados, levando até três horas para a entrega de uma máquina virtual.

Com a implementação de uma cultura de comunicação e colaboração entre as equipes de desenvolvimento e operações, e o uso da ferramenta Ansible, foi possível

diminuir o tempo de entrega da infraestrutura para quinze minutos. Além disso, reduziu os erros e possibilitou a reprodutibilidade do ambiente.

Soni (2015) realizou um estudo de caso da implementação de DevOps em uma empresa de seguros, da Índia. Segundo o autor, este é um mercado altamente competitivo, onde a velocidade na entrega de soluções de software podem fazer diferença, e ideias devem ser implementadas rapidamente. A empresa aplicou DevOps em sua operação, desde o processo de desenvolvimento do software até a infraestrutura para que o mesmo seja executado, com forte automação das rotinas. Entre as ferramentas utilizadas, o autor destaca o Jenkins para automação de *deploy* de software e o Chef para provisionamento da infraestrutura.

Utilizando as práticas de DevOps, além de outros benefícios, o autor relata que a entrega de software passou a ser muito mais rápida, com *bugs* sendo descobertos e corrigidos rapidamente. Com a automação da infraestrutura, conseguiu disponibilizar ambientes em dias, onde anteriormente era em semanas. Além disso, aumentou sua inovação, pois ideias começaram a entrar em prática continuamente.

Guerriero et al. (2019), entrevistam 86 empresas da Europa, e utilizaram os dados de 44 empresas que já utilizam infraestrutura como código, para descrever como está a adoção da tecnologia e os desafios que os profissionais encontram para adotá-la. O estudo apontou que as ferramentas utilizadas em IaC ainda possuem limitação, podendo estar em sua infância. Os profissionais ainda sentem a necessidade de novas técnicas para desenvolver, testar e manter a infraestrutura baseada em código. Entre os principais desafios, foram relatadas práticas recomendadas conflitantes, falta de testabilidade e problemas de legibilidade.

Punjabi e Bajaj (2016) realizam uma solução de ponta a ponta para testar as práticas e ferramentas de DevOps. Para isso, desenvolveram um pequeno software, que combinado com diversas ferramentas DevOps integradas, é disponibilizado ao usuário de modo automático, executando em um contêiner em nuvem. Entre as ferramentas utilizadas no estudo, estão o GIT, Chef, Docker e Jenkins.

Entre os benefícios identificados pelos autores, estão que devido aos testes automatizados de análise de código, erros e *bugs* foram detectados imediatamente. Pequenas alterações em código puderam ser disponibilizadas em minutos para os

usuários, e alterações maiores, em poucas horas. A conclusão foi que a abordagem DevOps pode resultar em entrega de software eficiente e indolor, permitindo que as organizações tenham seu foco orientado ao negócio, produzindo software de valor e com alcance rápido aos usuários.

O Quadro 1 apresenta as ferramentas utilizadas nos trabalhos relacionados. O trabalho de Guerriero et al. (2019), é uma pesquisa em diversas empresas sobre a adoção de IaC e, por este motivo, não é apresentado no quadro.

Quadro 1 - Ferramentas utilizadas nos trabalhos relacionados

	Rodrigues (2017)	Camões e Silva (2018)	Soni (2015)	Punjabi e Bajaj (2016)
Objetivo	Estudo de caso sobre a utilização do Puppet no gerenciamento de configurações	Estudo de caso na utilização do Ansible como ferramenta de implantação	Estudo de caso sobre a implementação de DevOps em uma empresa de seguros	Desenvolvimento de solução ponta a ponta para testar as práticas e ferramentas DevOps
Ferramenta de provisionamento	Não se aplica	Vagrant	Não se aplica	Não se aplica
Ferramenta de gerenciamento de configurações	Puppet	Ansible	Chef	Chef e Puppet
Ferramenta de Integração	Não se aplica	Jenkins	Jenkins	Jenkins
Ferramenta de automação de testes	Não se aplica	Não se aplica	Junit	Não se aplica
Ferramenta de gerenciamento de código fonte	Não se aplica	Não se aplica	SVN e Git	Git

Fonte: Elaborado pelo autor (2020).

4 MATERIAIS E MÉTODOS

Neste capítulo, serão expostos os procedimentos metodológicos adotados neste projeto que delinearam o caminho a ser seguido durante sua construção.

4.1 Tipo de pesquisa

Segundo Gil (2002) pesquisa é o ato de procurar, quando não existem informações suficientes para resolver um problema, respostas que possam resolvê-lo, utilizando para isso procedimentos racionais e sistemáticos.

Este trabalho pode ser considerado uma pesquisa aplicada, que conforme Marconi e Lakatos (2003), é dirigida a resolver problemas específicos, tendo como objetivo gerar conhecimento sobre o tema trabalhado para sua aplicação prática.

4.1.1 Definição da pesquisa quanto aos seus objetivos

Os objetivos de pesquisa deste trabalho enquadram-se em exploratórios, que segundo Botelho e da Cruz (2013), tem como um dos principais objetivos promover a familiaridade sobre o tema.

Estas pesquisas têm como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses. Pode-se dizer que estas pesquisas têm como objetivo principal o aprimoramento de ideias ou a descoberta de intuições (GIL, 2002, p. 41).

Desta forma, este trabalho é uma pesquisa exploratória, que tem como objetivo reunir informações e promover conhecimentos ao autor sobre o tema DevOps e Infraestrutura como Código, com a finalidade de aplicá-los em sua atuação profissional.

4.1.2 Definição da pesquisa quanto à natureza da abordagem

Este trabalho utilizou uma abordagem de pesquisa qualitativa, analisando os benefícios da adoção de uma Infraestrutura como Código em comparação com uma infraestrutura tradicional.

A pesquisa qualitativa é basicamente aquela que busca entender um fenômeno específico em profundidade. Ao invés de estatísticas, regras e outras generalizações, ela trabalha com descrições, comparações, interpretações e atribuição de significados possibilitando investigar valores, crenças, hábitos, atitudes e opiniões de indivíduos ou grupos. Permite que o pesquisador se aprofunde no estudo do fenômeno ao mesmo tempo em que tem o ambiente natural como a fonte direta para coleta de dados (BOTELHO; CRUZ, 2013, p. 54).

4.1.3 Definição da pesquisa quanto aos procedimentos técnicos

O procedimento técnico utilizado neste estudo pode ser classificado como pesquisa experimental.

Essencialmente, a pesquisa experimental consiste em determinar um objeto de estudo, selecionar as variáveis que seriam capazes de influenciá-lo, definir as formas de controle e de observação dos efeitos que a variável produz no objeto (GIL, 2002, p. 47).

Sendo assim, este trabalho explorou a metodologia DevOps com foco em Infraestrutura como Código, explorando e experimentando as ferramentas deste universo, onde o objetivo foi criar um ambiente totalmente definido como código.

4.2 Tecnologias

Para o desenvolvimento do projeto foram utilizadas ferramentas de IaC reconhecidas na área de TI por sua eficiência, que possuam documentação ampla e de qualidade e priorizando ferramentas livres para uso. Entre estas ferramentas, estão:

- a) **Ansible**: Ferramenta de gerenciamento de configurações de infraestrutura de TI, tais como: atualização do Sistema Operacional e instalação de softwares, de forma automatizada. Uma das suas principais características é a não necessidade de um agente no Sistema Operacional alvo, utilizando para sua conexão, ferramentas como o OpenSSH e o WinRM (ANSIBLE, 2020);
- b) **Django**: É um *framework* de desenvolvimento para a internet em Python, possuindo diversas ferramentas para tarefas comuns de desenvolvimento neste meio, como a autenticação de usuários e administração de conteúdo (DJANGO, 2020);
- c) **Docker**: É uma plataforma que permite a criação, execução e gerenciamento de contêineres. Um contêiner permite o empacotamento de um sistema operacional com todos os requisitos e configurações para a execução de uma determinada aplicação (DOCKER, 2020);
- d) **Github**: É um sistema de hospedagem, versionamento e gerenciamento de código fonte, baseado no Git (GITHUB, 2020);
- e) **Google Cloud Platform**: Plataforma de *cloud* do Google com infraestrutura global (GOOGLE CLOUD, 2020);
- f) **Kubernetes**: É um software de orquestração de contêineres. Com ele é possível implementar, dimensionar e gerenciar aplicativos em contêiner.

Entre seus principais recursos estão o balanceamento de carga, a automação de *rollouts*³ e *rollbacks*⁴ e o escalonamento horizontal (KUBERNETES, 2020);

- g) **Jenkins**: Software de integração e entrega contínuas. Conta com diversos *plugins* para suportar construção, implantação e automação de projetos de software (JENKINS, 2020);
- h) **PostgreSQL**: É um sistema de gerenciamento de banco de dados objeto-relacional, com mais de 30 anos de desenvolvimento ativo. É reconhecido por sua confiabilidade, robustez de recursos e desempenho (POSTGRESQL, 2020);
- i) **Terraform**: É uma ferramenta para provisionamento de infraestrutura em ambientes virtualizados, sejam estes locais ou em provedores de nuvem. Seu gerenciamento inclui instâncias de máquinas virtuais, armazenamento de dados e rede (TERRAFORM, 2020).

4.3 Desenvolvimento

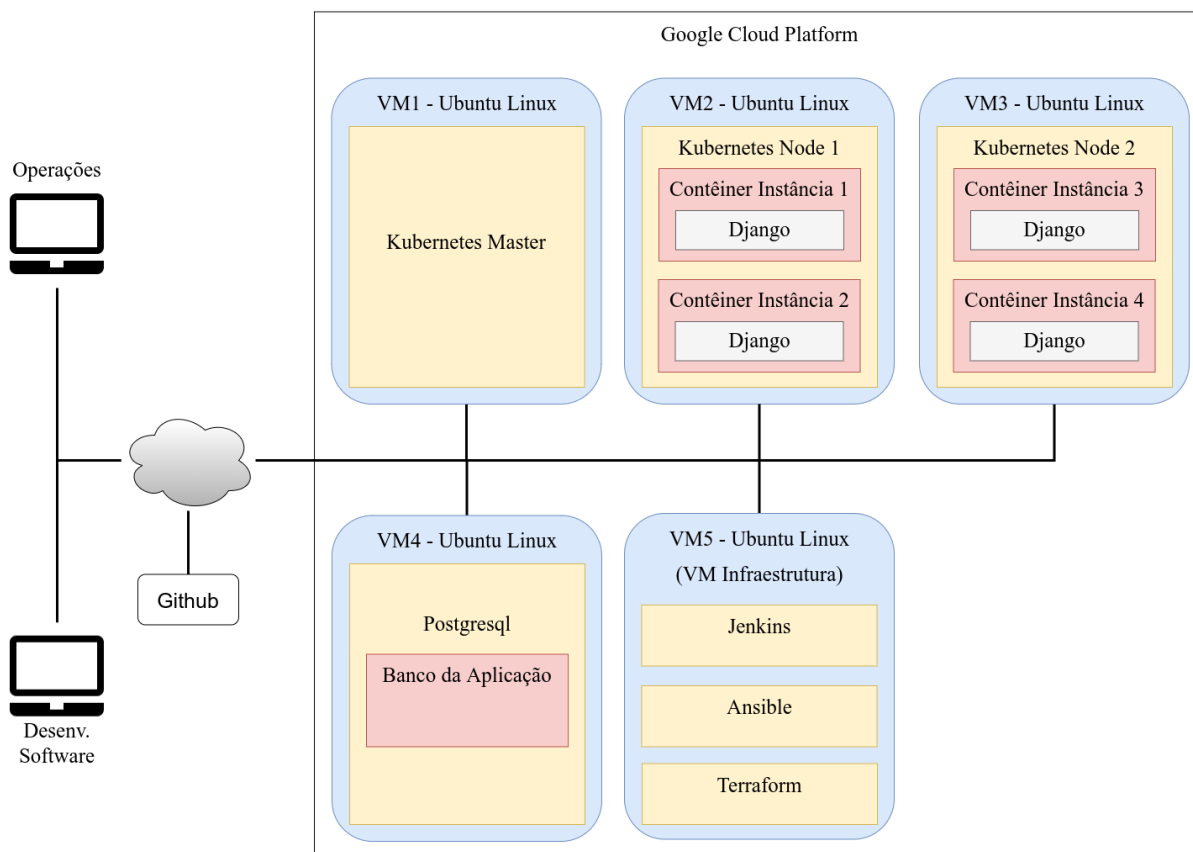
Este trabalho teve como objetivo a criação de uma infraestrutura completa, desde o início, com todos os recursos necessários para execução de um software em um *cluster* de contêineres hospedado em *cloud*, com a utilização de infraestrutura como código.

Para concepção da solução foi planejada uma arquitetura (FIGURA 4) que segmenta a infraestrutura em cinco máquinas virtuais hospedadas em nuvem.

³ *Rollout* quando relacionado a software, significa implementar um software ou uma atualização do mesmo.

⁴ *Rollback* quando relacionado a software, significa reverter uma determinada atualização de software, retornando ao seu estado anterior.

Figura 4 - Visão geral da arquitetura desenvolvida



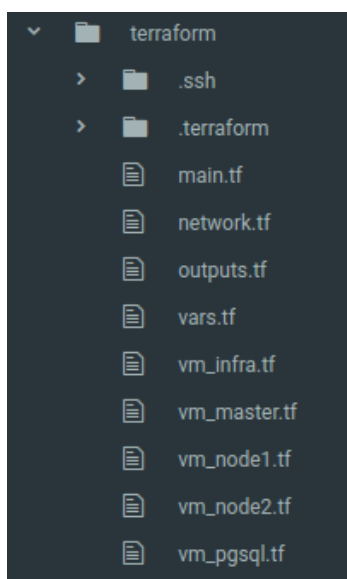
Fonte: Elaborado pelo autor (2020).

Na Figura 4 é exibido o *cluster* Kubernetes composto por um nó de gerenciamento e dois nós de trabalho, representado pelas VM1, VM2 e VM3. No Kubernetes cada VM pertencente ao *cluster* é chamada de nó, podendo ser estes de trabalho, que executam os contêineres e as tarefas envolvidas com suas execuções, ou de gerenciamento, que gerenciam o *cluster* e os outros nós. Neste *cluster* são executados contêineres com o *framework* Django e os serviços necessários para a execução da aplicação. A VM4 é responsável por executar o banco de dados da aplicação. Já a VM5 é utilizada pela equipe de operações para criar o ambiente, possuindo as ferramentas necessárias para criar a estrutura e realizar o *deploy* da aplicação. Na arquitetura também é exibida a ferramenta Github, utilizada para armazenamento do código da aplicação e da infraestrutura.

4.3.1 Provisionamento da infraestrutura em *cloud*

A criação das máquinas virtuais, configurações de rede e de *firewall* na plataforma de *cloud* do Google são realizadas com a utilização da ferramenta Terraform. Estas configurações na ferramenta foram realizadas localmente no computador do autor. O código de cada configuração foi versionado para o repositório Github, tornando-o acessível e executável de qualquer computador com acesso ao repositório. A estrutura de arquivos criada na ferramenta é demonstrada na Figura 5, onde a nomenclatura dos arquivos foi definida pelo autor.

Figura 5 - Estrutura de arquivos do Terraform



Fonte: Elaborado pelo autor (2020).

O diretório `.ssh` possui as chaves que serão instaladas pelo Terraform nas VMs criadas, para posteriormente permitir o acesso remoto para o usuário, administrador da rede ou outras ferramentas de IaC. Já o diretório `.terraform` possui arquivos de controle e binários utilizados pela ferramenta.

O arquivo `main.tf` (FIGURA 6) possui as configurações de conexão com o Google *Cloud*, como o apontamento para o arquivo com a credencial a ser utilizada, o nome do projeto e região geográfica do *datacenter* a ser utilizado. A autenticação com o Google *Cloud* é realizada através de uma conta de serviço criada anteriormente na plataforma, com permissões para criar e modificar os recursos do projeto.

O Terraform utiliza um arquivo de estado (chamado terraform.tfstate) que mantém documentado o estado atual da estrutura em nuvem, e que é indispensável para seu funcionamento. Como é uma boa prática e um requisito para o trabalho em equipe com a ferramenta, foi configurado no main.tf para este arquivo não ser armazenado no computador que está executando o Terraform, e sim na nuvem de forma segura.

Figura 6 - Terraform: arquivo main.tf

```
1 terraform {
2   backend "gcs" {
3     bucket    = "tcc-terraformstate"
4     prefix    = "tcc-infraascode"
5     credentials = "../uteis/Formatura-786e11b5710f.json"
6   }
7 }
8
9 provider "google" {
10   credentials = file("../uteis/Formatura-786e11b5710f.json")
11   project     = "formatura-249122"
12   region     = var.regiao
13 }
14
15 resource "random_id" "instance_id" {
16   byte_length = 8
17 }
```

Fonte: Elaborado pelo autor (2020).

No arquivo network.tf (FIGURA 7) são descritas as configurações de rede e *firewall* que devem ser criadas pelo Terraform. No arquivo, além dos endereços IPs privados a serem utilizados, são definidas as portas que estarão permitidas e visíveis na internet.

Figura 7 - Terraform: arquivo network.tf

```

1  resource "google_compute_network" "vpc" {
2      name                = "${var.projeto}-${var.env}-vpc"
3      auto_create_subnetworks = "false"
4      routing_mode         = "GLOBAL"
5  }
6
7  resource "google_compute_subnetwork" "private_subnet" {
8      name                = "${var.projeto}-${var.env}-pri-net"
9      ip_cidr_range       = var.rede_privada
10     network              = google_compute_network.vpc.self_link
11     region               = var.regiao
12 }
13
14 resource "google_compute_firewall" "allow-internal" {
15     name                = "${var.projeto}-fw-allow-internal"
16     network              = google_compute_network.vpc.name
17     allow {
18         protocol = "icmp"
19     }
20     allow {
21         protocol = "tcp"
22         ports    = ["0-65535"]
23     }
24     allow {
25         protocol = "udp"
26         ports    = ["0-65535"]
27     }
28     source_ranges = [
29         "${var.rede_privada}",
30     ]
31 }
32 resource "google_compute_firewall" "allow-http" {
33     name                = "${var.projeto}-fw-allow-http"
34     network              = google_compute_network.vpc.name
35     allow {
36         protocol = "tcp"
37         ports    = ["80", "443"]
38     }
39     target_tags = ["http"]
40 }
41 resource "google_compute_firewall" "allow-jenkins" {
42     name                = "${var.projeto}-fw-allow-jenkins"
43     network              = google_compute_network.vpc.name
44     allow {
45         protocol = "tcp"
46         ports    = ["8443", "8080"]
47     }
48     target_tags = ["jenkins"]
49 }
50 resource "google_compute_firewall" "allow-ssh" {
51     name                = "${var.projeto}-fw-allow-ssh"
52     network              = google_compute_network.vpc.name
53     allow {
54         protocol = "tcp"
55         ports    = ["22"]
56     }
57     target_tags = ["ssh"]
58 }

```

Fonte: Elaborado pelo autor (2020).

O arquivo `outputs.tf` possui informações que o Terraform deverá exibir após sua execução, como por exemplo, os endereços IPs das VMs criadas. Já no arquivo `vars.tf` (FIGURA 8), são definidas de variáveis customizadas que serão utilizadas em todos os arquivos do Terraform, facilitando uma eventual mudança ou até mesmo o reaproveitamento de código.

Figura 8 - Terraform: arquivo `vars.tf`

```
1 variable "env" { default = "dev" }
2 variable "projeto" { default = "tcc" }
3 variable "regiao" { default = "us-east1" }
4 variable "vm_master_zona" { default = "us-east1-d" }
5 variable "vm_node1_zona" { default = "us-east1-b" }
6 variable "vm_node2_zona" { default = "us-east1-c" }
7 variable "vm_pgsql1_zona" { default = "us-east1-c" }
8 variable "vm_infra_zona" { default = "us-east1-d" }
9 variable "nome_imagem" { default = "ubuntu-os-cloud/ubuntu-minimal-1804-lts" }
10 variable "tipo_vm" { default = "n1-highcpu-2" }
11 variable "rede_privada" { default = "10.139.0.0/24" }
```

Fonte: Elaborado pelo autor (2020).

Os arquivos `vm_infra.tf`, `vm_master.tf`, `vm_node1.tf`, `vm_node2.tf` e `vm_pgsql.tf` possuem as configurações de criação de VMs, como é exemplificado na Figura 9. Neste arquivo são definidas, além de outras configurações, o nome da VM, o tipo de *hardware* que ela deve utilizar, nome da imagem no Google, tamanho do disco e, através das *tags* de *firewall*, quais portas devem estar disponíveis para a internet. Neste arquivo também são definidos os usuários do sistema operacional a serem criados e as chaves de SSH que serão utilizadas em seus acessos, contidas no diretório `.ssh` do Terraform.

Figura 9 - Terraform: arquivo vm_infra.tf

```

1 resource "google_compute_instance" "vminfra" {
2   name           = "${var.projeto}-infra-${var.env}"
3   machine_type   = var.tipo_vm
4   zone           = var.vm_infra_zona
5   allow_stopping_for_update = true
6   tags           = ["ssh", "jenkins"]
7   boot_disk {
8     initialize_params {
9       image = var.nome_imagem
10      size  = 20
11    }
12  }
13  metadata = {
14    ssh-keys = "noll:${file(".ssh/id_rsa.pub")} \nubuntu:${file(".ssh/id_rsa.pub")}
15    *        \nubuntu:${file(".ssh/id_rsa_jenkins.pub")}"
16  }
17  network_interface {
18    subnetwork = google_compute_subnetwork.private_subnet.name
19    access_config {
20    }
21  }
22 }

```

Fonte: Elaborado pelo autor (2020).

A criação da estrutura é antecedida de uma etapa inicial, onde é criada a VM 5, no Terraform denominada VM Infraestrutura, e todas as configurações de rede necessárias para o seu funcionamento. Posteriormente, com a instalação do Jenkins e de todas as ferramentas necessárias para o gerenciamento da infraestrutura, é ele que será utilizado para disparar as rotinas de criação do restante da estrutura e *deploy* da aplicação. Este processo de execução parcial do Terraform é realizado passando parâmetros para que ele crie apenas esta parte da infraestrutura em *cloud* (FIGURA 10).

Figura 10 - Terraform: criação da VM infraestrutura

```

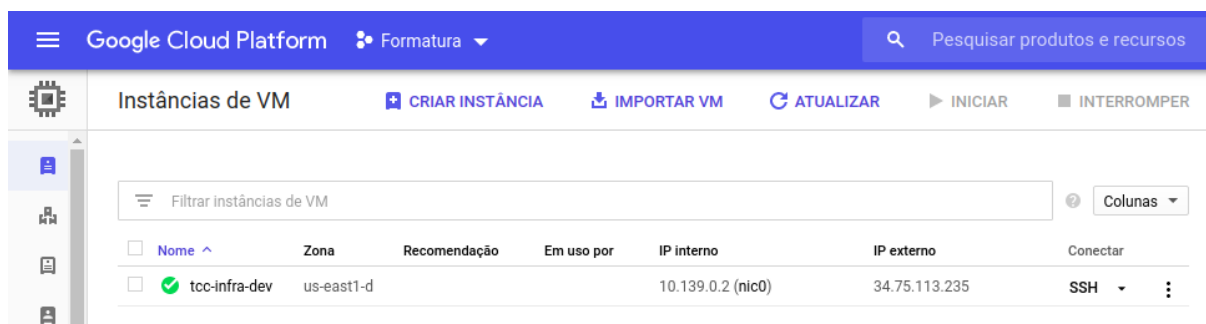
[noll:~/GIT/tcc-infraascode/terraform]$ terraform apply -target=google_compute_instance.vminfra
-target=google_compute_firewall.allow-jenkins -target=google_compute_firewall.allow-ssh

```

Fonte: Elaborado pelo autor (2020).

Após a execução do comando exibido na Figura 10, a VM Infra estará criada e em execução no Google *Cloud* (FIGURA 11), entretanto sem nenhuma configuração personalizada do Sistema Operacional e sem nenhum serviço em execução. Estas configurações ficam a cargo do Ansible.

Figura 11 - VM infraestrutura em execução



Nome	Zona	Recomendação	Em uso por	IP interno	IP externo	Conectar
tcc-infra-dev	us-east1-d			10.139.0.2 (nic0)	34.75.113.235	SSH

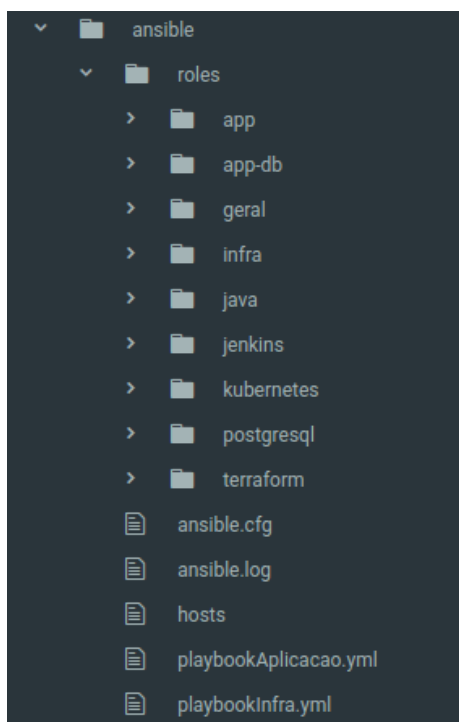
Fonte: Elaborado pelo autor (2020).

4.3.2 Gerenciamento de configurações

O gerenciamento de configurações da estrutura é realizada pelo Ansible, ficando ele responsável pela atualização e configuração do sistema operacional, bem como, a instalação e configuração de aplicativos. Neste projeto o Ansible exerce um papel fundamental, sendo ele responsável por instalar e configurar toda a estrutura após a criação das VMs pelo Terraform, finalizando sua execução com a estrutura concluída e com a aplicação de homologação sendo acessível pela internet.

A estrutura de arquivos do Ansible utilizada neste projeto é demonstrada pela Figura 12, onde são os arquivos *playbookAplicação.yml* (FIGURA 13) e *playbookInfra.yml* que definem as tarefas que devem ser executadas, em que ordem e em quais *hosts*. Os *playbooks* foram separados para que, assim como o Terraform, seja disponibilizada primeiramente a VM Infraestrutura, onde após a mesma estar em execução e com o Jenkins configurado, a criação do restante da estrutura é realizada pelo Jenkins.

Figura 12 - Estrutura de arquivos do Ansible



Fonte: Elaborado pelo autor (2020).

Figura 13 - Ansible: *Playbook* Aplicação

```
1 ---
2 - name: Playbook Aplicação - Configurar Sistema Operacional
3   hosts: servers
4   roles:
5     - geral
6
7 - name: Playbook Aplicação - Configurar Postgresql
8   hosts: dbs
9   roles:
10     - postgresql
11     - app-db
12
13 - name: Playbook Aplicação - Configurar Cluster Kubernetes
14   hosts: kubernetes
15   roles:
16     - kubernetes
17
18 - name: Playbook Aplicação - Configurar a Aplicação
19   hosts: masters
20   roles:
21     - app
22 ...
```

Fonte: Elaborado pelo autor (2020).

Nos *playbooks* são configuradas as *roles* que devem ser executadas em cada *host* ou grupo de *hosts*. *Roles* são conjuntos de tarefas a serem executadas pelo Ansible, que podem ser criadas manualmente ou através do *download* de *roles* disponibilizadas pela comunidade no portal Ansible Galaxy (<https://galaxy.ansible.com/>). O Quadro 2 demonstra a função e a origem das *roles* utilizadas neste projeto.

Quadro 2 - Ansible: Roles utilizadas

Nome da Role	Playbook	Função	Origem
app	Aplicação	Realiza o <i>deploy</i> da aplicação no <i>cluster</i> Kubernetes.	Autor
app-db	Aplicação	Importa o último backup do Postgresql armazenado no Github.	Autor
geral	Aplicação e Infra	Configurações gerais e atualização de pacotes do sistema operacional.	Autor
infra	Infra	Instalação de todos os requisitos para a execução do Ansible, docker e Django. Restore do último backup do Jenkins, importando todos os jobs criados anteriormente.	Autor
java	Infra	Instalação do java.	Ansible Galaxy
jenkins	Infra	Instalação do Jenkins e seus plugins.	Ansible Galaxy
kubernetes	Aplicação	Instalação e configuração do <i>cluster</i> Kubernetes.	Autor
postgresql	Aplicação	Instalação e configuração do Postgresql.	Ansible Galaxy
terraform	Infra	Instalação do binário do Terraform.	Autor

Fonte: Elaborado pelo autor (2020).

Uma *role* pode possuir diversas instruções em arquivos diferentes, chamadas de tarefas. Além disso, possui os diretórios *files* e *templates*, onde no primeiro são colocados arquivos que devem ser transmitidos para o sistema alvo, e no segundo *templates* de arquivos de configuração de aplicativos. A Figura 14 demonstra as tarefas da *role* App, que primeiramente configura a autenticação do *cluster* Kubernetes com o Google Cloud para que seja possível o *download* do contêiner da aplicação, e em seguida, realiza o *deploy* do mesmo e coloca a aplicação em execução.

Figura 14 - Ansible: Role App

```

1  ---
2  ### Chave para autenticacao no repositorio do google
3  - name: Adicionar a chave do google
4    template: src=kubernetes.json dest=/home/ubuntu force=yes owner=ubuntu
5    * group=ubuntu mode=0440
6
7  - name: Criar chave de autenticacao
8    become_user: ubuntu
9    shell: 'kubectl create secret docker-registry gcr-json-key --docker-
10    * server=eu.gcr.io --docker-username=_json_key --docker-password="$(cat ~/
11    * kubernetes.json)" --docker-email= && touch /var/ansible/
12    * cluster_gcpkey.txt'
13    args:
14    creates: /var/ansible/cluster_gcpkey.txt
15
16 - name: Patch serviceaccount
17 become_user: ubuntu
18 shell: 'kubectl patch serviceaccount default -p "{\"imagePullSecrets\":
19 * [{\"name\": \"gcr-json-key\"}]}" && touch /var/ansible/cluster_patchkey.txt'
20 args:
21 creates: /var/ansible/cluster_patchkey.txt
22
23 ### Subir a aplicacao
24 - name: Adicionando arquivo django.yaml
25 become_user: ubuntu
26 template: src=django.yaml dest=/home/ubuntu force=yes owner=ubuntu group=ubuntu
27 * mode=0644
28
29 - name: Criar deployment
30 become_user: ubuntu
31 shell: kubectl apply -f django.yaml && touch /var/ansible/cluster_deployment.txt
32 args:
33 creates: /var/ansible/cluster_deployment.txt
34
35 - name: Criar service
36 become_user: ubuntu
37 shell: kubectl expose deployment django-deployment --type=LoadBalancer --port 80
38 * --target-port 8000 --external-ip={{ansible_default_ipv4.address}} && touch /var/
39 * ansible/cluster_service.txt
40 args:
41 creates: /var/ansible/cluster_service.txt
42 ...

```

Fonte: Elaborado pelo autor (2020).

Os endereços IPs e nomes dos alvos do Ansible são configurados no arquivo *hosts* (FIGURA 15), onde as máquinas são agrupadas de acordo com suas funções. Além disso, neste arquivo também estão algumas variáveis utilizadas pela ferramenta, como o método de conexão com os *hosts* e a versão do Python que deve ser utilizada em sua execução.

Figura 15 - Ansible: arquivo *hosts*

```
1  [all:vars]
2  ansible_ssh_port=22
3  ansible_ssh_users=ubuntu
4  ansible_become=true
5  ansible_become_method=sudo
6  ansible_python_interpreter=/usr/bin/python3
7  master_hostname=tcc-kubemaster-dev
8
9  [masters]
10 master ansible_host=35.237.173.199
11
12 [nodes]
13 node1 ansible_host=34.75.191.105
14 node2 ansible_host=35.227.120.208
15
16 [dbs]
17 pgsql1 ansible_host=35.227.71.209
18
19 [infra]
20 infra1 ansible_host=34.75.113.235
21
22 [kubernetes:children]
23 masters
24 nodes
25
26 [servers:children]
27 masters
28 nodes
29 dbs
```

Fonte: Elaborado pelo autor (2020).

4.3.3 Automação com Jenkins

Neste projeto o Jenkins foi utilizado como um automatizador de rotinas, sendo ele responsável por criar e alterar toda a infraestrutura necessária para executar a aplicação, desde a criação das VMs com o Terraform, a configuração do ambiente com o Ansible e o *deploy* da aplicação.

Ao final da execução do `playbookInfra.yml` pelo Ansible, a VM Infraestrutura entra em execução com o Jenkins e todos os requisitos para gerenciar a estrutura e a aplicação. No processo de configuração do Ansible é restaurado o último *backup* do Jenkins, iniciando o mesmo já configurado com as rotinas necessárias (FIGURA 16).

Figura 16 - Página inicial do Jenkins

The screenshot shows the Jenkins web interface. At the top, there's a navigation bar with the Jenkins logo, a search bar, and user profile icons. Below this, a sidebar on the left contains links for 'Novo job', 'Usuários', 'Histórico de compilações', 'Gerenciar Jenkins', 'Minhas views', 'Credentials', and 'New View'. The main area displays a table of jobs under the 'Tudo' tab. The table has columns for status (S, W), name, last success, last failure, and last duration. Below the table, there are sections for 'Fila de builds' (empty) and 'Estado do executor de builds' (showing 1 Parado and 2 Parado). At the bottom, there are links for 'Ícone: S M L', 'Legenda', and three Atom feed links.

S	W	Nome ↓	Último sucesso	Última falha	Última duração
		Atualizar Sistema de Notas	N/D	N/D	N/D
		Backup Jenkins	N/D	N/D	N/D
		Backup PostgreSQL	N/D	N/D	N/D
		Criar ou Atualizar Infraestrutura	N/D	N/D	N/D
		Remover Infraestrutura	N/D	N/D	N/D

Ícone: [S](#) [M](#) [L](#) [Legenda](#) [Atom feed de tudo](#) [Atom feed das falhas](#) [Atom feed apenas para os últimos builds](#)

Fonte: Elaborado pelo autor (2020).

Na primeira execução do `playbookInfra.yml` pelo Ansible, foi disponibilizada uma instalação padrão do Jenkins, configurada com o usuário de acesso, plugins instalados e com rotina de backup, porém sem tarefas configuradas. As tarefas foram configuradas utilizando a própria interface da ferramenta, que versiona as alterações de suas configurações com o Github. Desta forma, ao refazer o ambiente é realizado o restore da última versão das configurações do Jenkins, iniciando o mesmo com suas tarefas atualizadas.

A rotina “Atualizar Sistema de Notas” é responsável por realizar o *download* da última versão da aplicação no Github e realizar o seu *deploy*. O processo de *deploy* foi dividido em três blocos de comando em *shell* do Jenkins (FIGURA 17), onde se alguma das etapas falharem, a rotina é interrompida e a aplicação continua sua execução na versão estável e sem sofrer interrupções:

- 1º bloco *shell*: realiza os testes que garantam o funcionamento da aplicação;
- 2º bloco *shell*: realiza a atualização do banco de dados da aplicação;
- 3º bloco *shell*: realiza a criação de um novo contêiner atualizado com a aplicação e o *rollout* do mesmo no *cluster* Kubernetes.

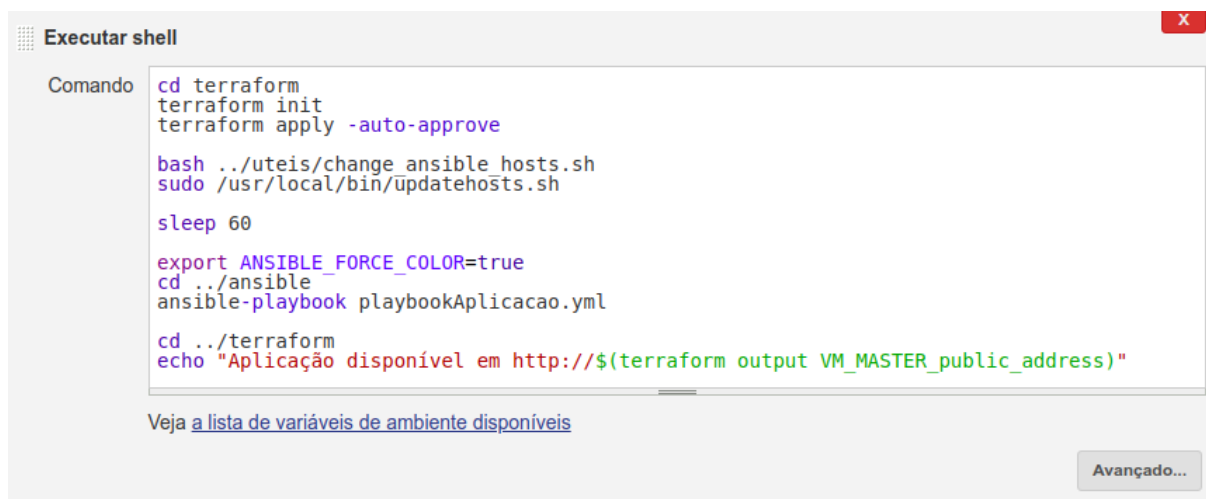
Figura 17 - Jenkins: atualizar sistema de notas



Fonte: Elaborado pelo autor (2020).

A rotina “Criar ou Atualizar Infraestrutura” é responsável pela criação ou atualização da infraestrutura da aplicação, criando e configurando as VMs e os softwares necessários. A rotina inicia com a execução do Terraform para a criação das VMs e na sequência executa dois scripts para atualizar o arquivo *hosts* do Ansible com os novos endereços IPs das VMs e também o arquivo */etc/hosts* da VM Infraestrutura. Na sequência é executado o *playbookAplicacao.yml* do Ansible que realiza a configuração de todo o ambiente, disponibilizando ao término a aplicação em execução. Os comandos de execução da rotina são exibidos na Figura 18.

Figura 18 - Jenkins: criar ou atualizar infraestrutura

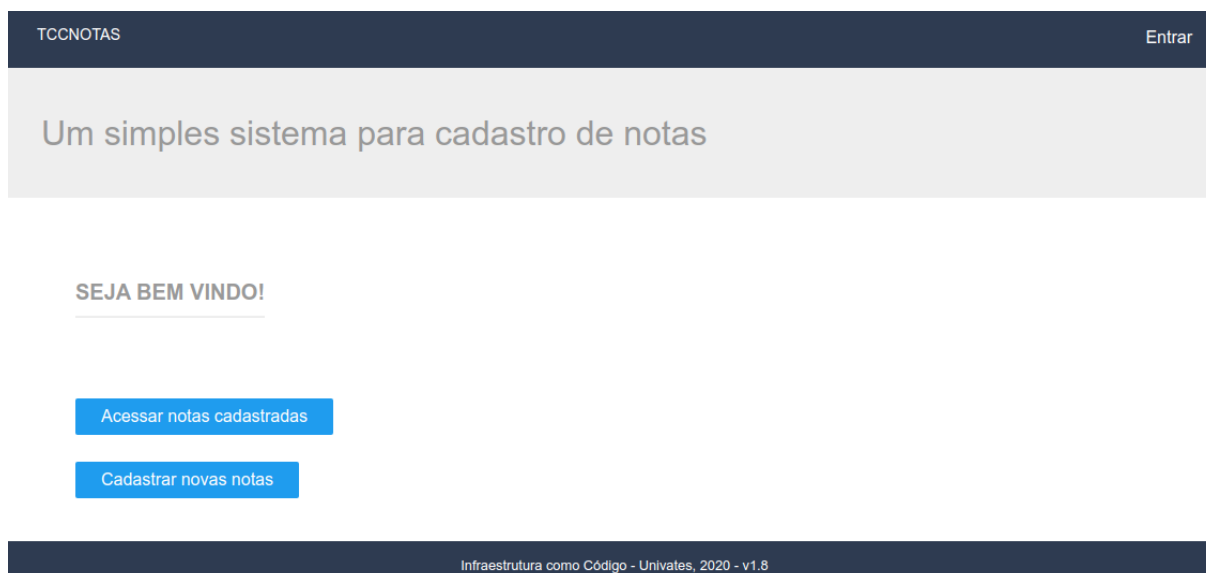


Fonte: Elaborado pelo autor (2020).

4.3.4 Aplicação para homologar infraestrutura

Para realizar testes e validar a infraestrutura desenvolvida, exibindo ao final da criação e configuração da mesma a execução de uma aplicação com acesso ao banco de dados, foi realizado o desenvolvimento de uma pequena aplicação de exemplo, possuindo como função o cadastro de notas de texto, com a ideia de lembretes, chamada TCCNOTAS (FIGURA 19).

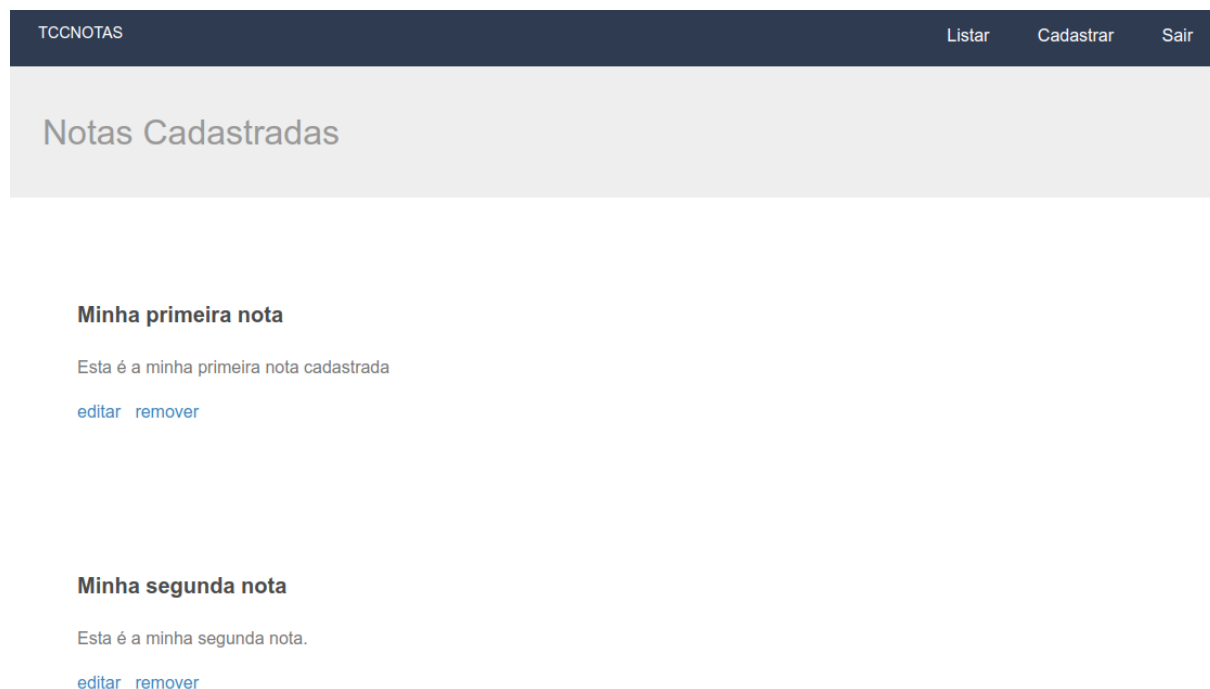
Figura 19 - Aplicação: tela inicial



Fonte: Elaborado pelo autor (2020).

Ao realizar o *login* na aplicação, são exibidas as notas cadastradas pelo usuário, conforme Figura 20. No menu superior há a opção do usuário cadastrar uma nova nota, informando um nome e descrição para a mesma (FIGURA 21).

Figura 20 - Aplicação: listar notas



Fonte: Elaborado pelo autor (2020).

Figura 21 - Aplicação: cadastrar nota

TCCNOTAS Listar Cadastrar Sair

Cadastrar Nota

Cadastro

Nome:

Descrição:

Infraestrutura como Código - Univates, 2020 - v1.8

Fonte: Elaborado pelo autor (2020).

A aplicação é baseada em Python para Web, sendo utilizado para seu desenvolvimento o *framework* Django. Além de ser utilizada como a aplicação que valida a infraestrutura, estando em funcionamento após a execução de todas as ferramentas de IaC, ela também foi importante para demonstrar algumas práticas de DevOps, como os testes automatizados, o versionamento do banco de dados e o *deploy* automatizado e sem queda da aplicação.

Para realizar a configuração completa do ambiente e disponibilizar a aplicação em funcionamento, durante o processo de execução das ferramentas de IaC é restaurado o último *backup* válido do banco de dados. Para isso, foi utilizado como repositório do *backup* o Github, sendo o mesmo atualizado a cada 30 minutos quando a aplicação está em execução. É importante ressaltar que esta é uma aplicação com banco de dados mínimo, praticamente sem dados, onde a estratégia de *backup* e *restore* para aplicações maiores devem ser definidas de acordo com as necessidades e particularidades das mesmas.

4.4 Testes e análise dos resultados

Como já descrito anteriormente, a execução das tarefas de IaC neste projeto foram divididas em duas etapas, que são a criação e configuração da VM Infraestrutura e a criação e configuração da infraestrutura da aplicação. Para a criação e configuração da VM Infraestrutura (QUADRO 3), em uma média de cinco execuções, são necessários 14 minutos e 43 segundos. Sendo 54 segundos para o Terraform criar a VM e as configurações de rede no Google *Cloud* e mais 13 minutos e 48 segundos para o Ansible realizar a configuração e instalação de aplicativos.

Quadro 3 - Tempos de execução para VM infraestrutura

Ferramenta		Número de tarefas	Tempo médio (minutos)
Terraform		5	00:54
Ansible	Role		
	geral	5	01:24
	java	9	01:31
	jenkins	60	06:00
	infra	19	04:19
	terraform	6	00:34
Total		104	14:43

Fonte: Elaborado pelo autor (2020).

Nesta etapa é possível visualizar um tempo muito maior de execução do Ansible, isso se dá porque o Terraform é responsável apenas por criar as VMs e configurações de rede no provedor de *Cloud*, e o Ansible é responsável por atividades que consomem maior tempo, como as atualizações do sistema operacional e instalar e configurar aplicativos.

As *roles* do Ansible que consumiram maior tempo foram a *jenkins* e *infra*, responsáveis pela instalação e configuração do Jenkins e de todas as aplicações necessárias para criar a infraestrutura da aplicação e realizar o seu *deploy*. No Quadro 4 são exibidas as 10 tarefas com maior tempo de execução, ficando evidente que as atividades mais demoradas são aquelas relacionadas a instalação de aplicativos, onde é necessário o *download* dos mesmos.

Quadro 4 - Tarefas com maior tempo de execução para VM infraestrutura

Tarefa	Tempo médio (segundos)
jenkins : Install Jenkins plugins	52
java : Install OpenJDK Java 8	39
infra : Instalar <i>Cloud</i> SDK	33
infra : Instalar Docker	24
infra : Instalar Django e psycpg2	22
geral : Instalar pacotes	21
infra : Instalar ansible	21
geral : Alteracao do <i>/etc/profile</i>	18
geral : Atualizar Sistema Operacional	18
jenkins : Wait for Jenkins	17

Fonte: Elaborado pelo autor (2020).

O equipamento utilizado nestas execuções foi o computador do autor, demonstrado pelo Quadro 5.

Quadro 5 - Hardware de execução para VM infraestrutura

Equipamento:	Notebook Dell Inspiron 7000
Processador:	Intel Core i5 2.5GHz (4 cores)
Memória:	8GB
Disco Rígido:	SSD
Sistema Operacional:	Ubuntu 18.04.04 LTS
Versão do Ansible:	2.9.9
Versão do Terraform:	0.12.26

Fonte: Elaborado pelo autor (2020).

Para criar o restante da infraestrutura, ou seja, a infraestrutura da aplicação, foram necessários apenas 3 minutos e 49 segundos (QUADRO 6), na média de cinco execuções. Este tempo muito inferior se comparado ao tempo de execução para a criação da VM Infraestrutura, se deve ao fato da execução ser originada no próprio *Google Cloud*. Na execução anterior o processo é realizado pela internet, uma vez que não há nenhuma estrutura criada até então no *Google Cloud*.

Quadro 6 - Tempos de execução para infraestrutura da aplicação

Ferramenta	Número de tarefas	Tempo médio (minutos)
Terraform	7	00:19
Scripts de ajustes de IPs	2	00:03
Ansible	Role:	
	geral	00:44
	postgresql	00:31
	app-db	00:08
	kubernetes	01:57
	app	00:06
Total	73	03:49

Fonte: Elaborado pelo autor (2020).

A *role* do Ansible que possuiu maior tempo de execução foi a *kubernetes*, principalmente por realizar a instalação de diversos pacotes, a configuração e inicialização do *cluster*. No Quadro 7 são exibidas as 10 tarefas com maior tempo de execução, onde a maioria são pertencentes a *role* Kubernetes.

Quadro 7 - Tarefas com maior tempo de execução para infraestrutura da aplicação

Tarefa	Tempo médio (segundos)
kubernetes : Inicializar o <i>cluster</i>	43
geral : Instalar pacotes	20
kubernetes : Instalar Docker	18
geral : Atualizar Sistema Operacional	15
postgresql : Ensure PostgreSQL packages are installed	15
kubernetes : Entrar no <i>cluster</i>	13
kubernetes : Instalar kubelet	11
kubernetes : Instalar kubeadm	8
kubernetes : Adicionar repositório APT Kubernetes	6
postgresql : Ensure PostgreSQL Python libraries are installed	3

Fonte: Elaborado pelo autor (2020).

A estrutura de hardware e software utilizadas nas execuções da infraestrutura da aplicação são demonstradas pelo Quadro 8. Esta etapa da infraestrutura foi executada diretamente do Jenkins, instalado e configurado na VM Infraestrutura criada anteriormente e hospedada no Google *Cloud*.

Quadro 8 - Hardware de execução para infraestrutura da aplicação

Equipamento:	VM Infraestrutura no Google <i>Cloud</i>
Processador:	Intel Xeon 2.30GHz (2 cores)
Memória:	1,78GB
Disco Rígido:	SSD
Sistema Operacional:	Ubuntu 18.04.04 LTS
Versão do Ansible:	2.9.9
Versão do Jenkins:	2.222.4
Versão do Terraform:	0.12.24

Fonte: Elaborado pelo autor (2020).

Somados os tempos de execução das duas etapas, a infraestrutura é criada e o sistema disponibilizado em execução com uma média de 18 minutos e 32 segundos. Este também poderia ser considerado o tempo necessário para duplicar a estrutura em uma estrutura de testes ou desenvolvimento, bastando apenas alterar algumas variáveis do Terraform e o apontamento de IPs no Ansible. O tempo de uma recuperação de desastres também fica muito próximo a este, adicionando a ele apenas

as etapas de configuração de uma nova conta no Google *Cloud* e das permissões para conexão do Terraform, com tempo adicional, alterando variáveis do Terraform para outro provedor de *Cloud* suportado pela ferramenta.

O tempo estimado na criação de todos os códigos envolvidos na infraestrutura desenvolvida neste trabalho, sem conhecimento prévio do autor com as ferramentas de IaC, foi de aproximadamente 150 horas. Neste período, foram realizados 110 *commits*⁵ na ferramenta de versionamento de código e a infraestrutura foi destruída e reconstruída 69 vezes no Google *Cloud*.

O Quadro 9 apresenta a quantidade de arquivos de código ou configuração gerados no trabalho, totalizando 151 arquivos divididos pelas ferramentas Terraform e Ansible, que são as ferramentas responsáveis pela criação e configuração de todo o ambiente. Arquivos utilizados por outras ferramentas, como por exemplo o Jenkins, estão incluídos em sua *role* correspondente.

Quadro 9 - Número de arquivos gerados

Ferramenta	Função/Tarefa	Número de arquivos
Terraform	Configurações gerais	3
	Criar VMs em <i>Cloud</i>	8
	Configurações de rede e <i>firewall</i>	1
Ansible	Configurações gerais	4
	Role app	4
	Role app-db	6
	Role geral	4
	Role infra	9
	Role java	5
	Role jenkins	76
	Role kubernetes	5
	Role postgresql	23
	Role terraform	3
Total		151

Fonte: Elaborado pelo autor (2020).

Todos os arquivos utilizados neste trabalho foram versionados, com histórico de alterações. Caso necessário reverter alguma configuração, é possível fazer uso dos recursos de versionamento, já conhecidos e utilizados no desenvolvimento de software. Com isso, a utilização de IaC trouxe histórico, rastreabilidade, documentação

⁵ Envio de atualização de código para a ferramenta de versionamento de código.

e organização, onde cada ferramenta possui seus arquivos e configurações separados, com suas atribuições bem definidas.

Com as ferramentas de IaC foi possível realizar todos os procedimentos necessários para criar e configurar a infraestrutura proposta, desde a criação das VMs, a configuração do sistema operacional, a configuração de um *cluster* Kubernetes, e até mesmo, o *deploy* da aplicação. É notável o quanto a infraestrutura definida como código fica facilmente escalável, tornando possível fazer em minutos o que na infraestrutura tradicional poderia levar horas. Além disso, resolve a dependência com os provedores de *Cloud*, uma vez que torna possível com poucas alterações de código o seu provisionamento em outra estrutura, sendo também considerado um ótimo processo de recuperação de desastres.

5 CONSIDERAÇÕES FINAIS

Cada vez mais a tecnologia faz parte do dia-a-dia das pessoas, onde a demanda por sistemas cresce de forma acelerada. Para atender esta demanda, foram criadas diversas metodologias de desenvolvimento de software com agilidade, entretanto a infraestrutura tradicional não acompanhou em mesmo ritmo. Além disso, os conflitos internos entre as equipes de desenvolvimento e operações prejudicam ainda mais o processo de disponibilização de um software ou de sua atualização.

A utilização da metodologia DevOps, torna possível que as equipes de desenvolvimento e operações trabalhem juntas, apoiando-se e complementando-se. Com o uso de ferramentas de automação da infraestrutura, torna-se a mesma escalável e dinâmica, atendendo com velocidade as necessidades de suas aplicações.

Entre os diversos benefícios da adoção da infraestrutura como código, com este trabalho foi possível evidenciar a recuperação de desastres. Uma vez que a infraestrutura completa está em formato de código, basta alterar apontamos do provedor de *Cloud* para em alguns poucos minutos a estrutura estar em execução novamente em outros servidores. Isso também acaba trazendo a independência do provedor de *Cloud*, onde no cenário tradicional uma migração de provedor acaba sendo demorada, trabalhosa e, por muitas vezes, traumática.

Ao utilizar a infraestrutura baseada em código, a documentação básica do ambiente é automática, uma vez que o próprio código também é documentação. Além disso, manter o código versionado garante que alterações mal sucedidas não causem um desastre, havendo sempre a possibilidade de retornar a infraestrutura para um

estado conhecido e funcional. Para isso, também é fundamental a repetitividade de execução das ferramentas de IaC, pois somente assim serão evitadas alterações de forma manual no ambiente, onde na próxima execução se não forem incluídas como código, serão removidas.

A infraestrutura como código ainda é pouco difundida, principalmente em nossa região. Espera-se que este trabalho levante o interesse e discussões a respeito do tema, levando profissionais da área de operações a realizar experimentos e futuras implantações.

REFERÊNCIAS

AMAZON. **O que é DevOps?**. 2019. Disponível em:
<<https://aws.amazon.com/pt/devops/what-is-devops/>>. Acesso em: 08 out. 2019.

ANSIBLE. **Why Ansible?**. 2020. Disponível em:
<<https://www.ansible.com/overview/it-automation>>. Acesso em: 25 maio 2020.

ARTAČ, Matej; BOROVŠAK, Tadej; DI NITTO, Elizabetta; GUERRIERO, Michele; TAMBURRI, Damian A. DevOps: Introducing Infrastructure-as-Code. In: International Conference on Software Engineering Companion, 39., 24 ago. 2017. **Anais...** Buenos Aires: IEEEExplore, 2017. Disponível em:
<<https://ieeexplore.ieee.org/abstract/document/7965401>>. Acesso em: 02 out. 2019.

BEYER, Betsy; JONES, Chris; PETOFF, Jennifer; MURPHY, Niall R. **Engenharia de Confiabilidade do Google**: Como o Google administra seus sistemas de produção. São Paulo: Novatec, 2016.

BOTELHO, Joacy M.; CRUZ, Vilma A. G. **Metodologia Científica**. São Paulo: Pearson Education do Brasil, 2013.

CAMÕES, Renato J. S.; SILVA, Jessé A. Desenvolvimento e Operações utilizando Ansible como principal ferramenta de implantação: Estudo de caso no Tribunal de Justiça do Distrito Federal (TJDFT). **Revista Tecnologias em Projeção**, v. 9, n. 2, 36-52, 2018. Disponível em:
<<http://revista.faculdadeprojecao.edu.br/index.php/Projecao4/article/view/1140/956>>. Acesso em: 02 out. 2019.

DJANGO. **Why Django?**. 2020. Disponível em: <<https://www.djangoproject.com/start/overview/>>. Acesso em: 27 maio 2020.

DOCKER. **Developers Bring Their Ideas to Life With Docker**. 2020. Disponível em: <<https://www.docker.com/why-docker>>. Acesso em: 27 maio 2020.

EDWARDS, Damon. What is DevOps?. **dev2ops.org**, 2010. Disponível em: <<http://dev2ops.org/2010/02/what-is-devops/>>. Acesso em: 23 set. 2019.

GEERLING, Jeff. **Ansible for DevOps**: Server and configuration management for humans. Canadá: Leanpub, 2015.

GIL, Antonio C. **Como Elaborar Projetos de Pesquisa**. 6. ed. São Paulo: Atlas, 2002.

GITHUB. **How Developers Work**. 2020. Disponível em: <<https://github.com/features>>. Acesso em: 26 maio 2020.

GOOGLE CLOUD. **Visão Geral do Google Cloud Platform**. 2020. Disponível em: <<https://cloud.google.com/docs/overview>>. Acesso em: 27 maio 2020.

GUERRIERO, Michele; GARRIGA, Martin; TAMBURRI, Damian A.; PALOMBA, Fabio. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In: International Conference on Software Maintenance and Evolution, 30 set. a 03 out. 2019. **Anais...** Ohio: ICSME, 2019. 10p. Disponível em: <<https://dibt.unimol.it/staff/fpalomba/documents/C47.pdf>> Acesso em: 03 out. 2019.

HEWLETT PACKARD ENTERPRISE. **O que é Infraestrutura como Código?**. 2019. Disponível em: <<https://www.hpe.com/br/pt/what-is/infrastructure-as-code.html>>. Acesso em: 12 de out. 2019.

HUMBLE, Jez; FARLEY, David. **Continuous Delivery**: Reliable software releases through build, test, and deployment automation. Boston: Pearson Education, 2010.

HÜTTERMANN, Michael. **DevOps for Developers**: Integrate development and operations, the agile way. New York: Apress, 2012.

INSTRUCT. **Guia definitivo de Infraestrutura Ágil**. 2017. Disponível em: <<https://instruct.com.br/wp-content/uploads/2017/05/guia-completo-infra-agil.pdf>>. Acesso em: 15 out. 2019.

JENKINS. **Jenkins User Documentation**. 2020. Disponível em: <<https://www.jenkins.io/doc/>>. Acesso em: 26 maio 2020.

KIM, Gene; HUMBLE, Jez; DEBOIS, Patrick; WILLIS, John. **Manual de DevOps**: Como obter agilidade, confiabilidade e segurança em organizações tecnológicas. Rio de Janeiro: Alta Books, 2018.

KUBERNETES. **Kubernetes Features**. 2020. Disponível em: <<https://kubernetes.io/>>. Acesso em: 24 maio 2020.

MARCONI, Marina A.; LAKATOS, Eva M. **Fundamentos de metodologia científica**. 5. ed. São Paulo: Atlas, 2003.

MORRIS, Kief. **Infrastructure as Code**: Managing server in the cloud. Sebastopol: O'Reilly Media, 2016.

NELSON-SMITH, Stephen. **Test-Driven Infrastructure with Chef**: Bring behavior-driven development to infrastructure as code. Sebastopol: O'Reilly Media, 2013.

POSTGRESQL. **About PostgreSQL**. PostgreSQL, 2020. Disponível em: <<https://www.postgresql.org/about/>>. Acesso em: 25 maio 2020.

PUNJABI, Rahul; BAJAJ, Ruhi. **User Stories to User Reality**: A DevOps Approach for the Cloud. Bangalore: IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), 2016. Disponível em: <<https://ieeexplore.ieee.org/abstract/document/7807905>>. Acesso em: 06 out. 2019.

RITI, Pierluigi. **Pro DevOps with Google Cloud Platform**: With Docker, Jenkins, and Kubernetes. New York: Apress, 2018.

RODRIGUES, Carlos H. B. **Gerenciamento de configurações com Puppet**: Garantindo uma Infraestrutura ágil, estável e segura. 2017. 54 f. Monografia (Especialização em Rede de Computadores com ênfase em Segurança) – Centro Universitário de Brasília, Brasília, 2017. Disponível em: <<https://repositorio.uniceub.br/jspui/handle/235/12406>>. Acesso em: 02 out. 2019.

SAITO, Hideto; LEE, Hui-Chuan Chloe; WU, Cheng-Yang. **DevOps With Kubernetes**: Accelerating software delivery with container orchestrators. Birmingham: Packt Publishing, 2017.

SATO, Danilo. **DevOps**: Na prática: Entrega de software confiável e automatizada. São Paulo: Casa do Código, 2014.

SHAHER, Andrew C. **Agile Infrastructure**: A Story in Three Acts. San Jose: Velocity, 2009.

SHARMA, Sanjeev. **DevOps For Dummies**: IBM Limited Edition. New Jersey: John Wiley & Sons, 2014.

SOMMERVILLE, Ian. **Engenharia de Software**. 9a edição. São Paulo: Pearson Prentice Hall, 2011.

SONI, Mitesh. **End to End Automation On Cloud with Build Pipeline**: The case for DevOps in Insurance Industry. In: International Conference on Cloud Computing in Emerging Markets, 25 a 27 nov. 2015. **Anais...** Bangalore: CCEM, 2015. Disponível em: <<https://ieeexplore.ieee.org/abstract/document/7436936>>. Acesso em: 04 out. 2019.

TERRAFORM. **Introduction to Terraform**. 2020. Disponível em: <<https://www.terraform.io/intro/index.html>>. Acesso em: 26 maio 2020.

VADAPALLI, Sricharan. **DevOps: Continuous Delivery, Integration, and Deployment with DevOps**: Rapid Learning Solution. Birmingham: Packt, 2018.

VENTAPANE, Dennis. Cultura DevOps: entenda o que é quais os seus benefícios. **Profissionais TI**, 2019. Disponível em: <<https://www.profissionaisiti.com.br/2019/06/cultura-devops-entenda-o-que-e-quais-os-seus-beneficios/>>. Acesso em: 06 out. 2019.



UNIVATES

R. Avelino Tallini, 171 | Bairro Universitário | Lajeado | RS | Brasil
CEP 95900.000 | Cx. Postal 155 | Fone: (51) 3714.7000
www.univates.br | 0800 7 07 08 09